字符串理论

西安交通大学
张博航
2017.7.5

第一节 单模式匹配

1.1 字符串

定义1：基本术语
（1）字母表：用∑表示，字母表大小记为|∑|；
（2）字母表上的字符：用小写字母a,b,c等表示；
（3）字符串：用大写字母S,T,W,X,Y,Z等表示；
（4）字符串的长度：|S|。
（5）空串：用ε表示，空串长度为0；
（6）字符串的连接：记为ST；

定义2：
（1）前缀：设S=XY，Y可空，称X是S的前缀；
（2）后缀：设S=XY，X可空，称Y是S的后缀；
（3）子串：设S=XWY，X,Y可空，称W是S的子串；
将S中从位置i到位置j这一子串记为S[i,j]。

定义3：匹配
若W是S的子串，称W匹配了S，称S为主串或文本（串），W为子串或模式（串）。

例1：字符串abaababa被模式a匹配了5次，被模式aba匹配了3次。字符串aaaaaaa被aaaa匹配了4次。

1.2 KMP算法引入

朴素匹配算法

a	a	b	a	b	a	a	b	a	b	b

a	b	a	b	b

时间复杂度O(n^2)

a	a	a	a	a	a	a	a	a	a	b

a	a	a	a	b

KMP算法

b	a	a	b	a	b	a	a	b	a	b	b

a	b	a	b	b

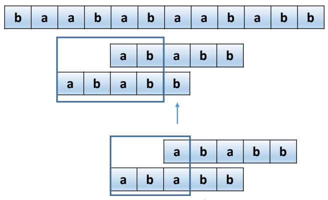
↑

可以看出，充分利用已匹配的信息，模式串可以跳跃地向右移动。每次模式串移动后，指针之前内容均被匹配。每次操作，要么指针向右移动，要么模式串向右移动。且指针总是向右移动|S|次，模式串向右移动不超过|S|次，这意味着算法时间复杂度可以达到线性。

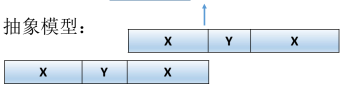
第一节 单模式匹配

ACM XJTU

问题的关键在于模式串向右滑动多少！



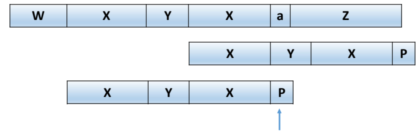
抽象模型：



7

第一节 单模式匹配

ACM XJTU



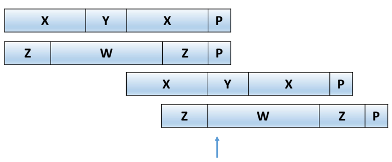
也就是在模式串中找到子串X，分别在开头和结尾出现！
满足该条件当且仅当匹配指针之前的串！

若有多个这样的X，显然应选择长度最大的一个。
这是因为，根据朴素算法的思想，模式串每次只向右滑动1。而KMP算法实质上只是过滤了不满足要求的情况，故仍应滑动的尽可能少，自然得证。

8

第一节 单模式匹配

ACM XJTU



选择长度最大的一个X将模式串滑动后，若不匹配指针位置，则需要进一步滑动到长度较小的位置。于是我们就这样不断滑动，直到指针位置匹配为止。这就是KMP的基本思想。
可以发现，最大X的选取只和模式串有关，和主串无关，它是模式串的固有性质。我们有必要加以研究。

9

第一节 单模式匹配

ACM XJTU

1.3 border、fail及其性质

定义4：border和Border
设X≠S满足X是S的前缀，且是S的后缀，则称X是S的一个border。S的所有border之集记作border(S)。border(S)中长度最大的X记作Border(S)。

定义5：fail数组
设|S|=n，则fail数组是一个有n个元素的数组，其中fail[i]=|Border(S[0,i])|，规定fail[0]=0。显然fail[i]≤i。

故之前所说的“最大X”实际上就是Border。

10

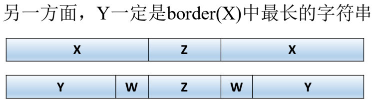
第一节 单模式匹配

ACM XJTU

定理1：border(S)中所有字符串的长度可表示为：
fail[|S|-1],fail[fail[|S|-1]-1],……,0。

证明：首先，|Border(S)|=fail[|S|-1]。
将border(S)中所有字符串X从大到小排序，设X和Y为排序后相邻两字符串，下证|Y|=fail[|X|-1]。

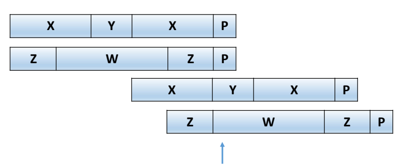
一方面，Y是border(X)；
另一方面，Y一定是border(X)中最长的字符串。



11

第一节 单模式匹配

ACM XJTU



当指针位置不匹配时，需要滑动模式串。由定理1知，滑动过程中，匹配长度依次为（设初始匹配串为S）：
fail[|S|-1], fail[fail[|S|-1]-1],……,0。

假设我们能求出fail数组（对i从0到|S|-1）,那么就得到如下的KMP算法。

12

第一节 单模式匹配

ACM XJTU

算法1：KMP算法

b	a	a	b	a	b	a	a	b	a	b	b

a	b	a	b	b

i	fail[i]
0	0
1	0
2	1
3	2
4	0

初始化：匹配长度j=0，指针i=0；
重复以下步骤，直到指针i达到主串末尾：
（1）若当前指针位置匹配，则i++；
此时若j达到模式串末尾，则匹配成功！
（2）否则，重复j=fail[j-1]，直到当前指针位置匹配；
或者j=0。

13

第一节 单模式匹配

ACM XJTU

算法1：KMP算法

```
#define MAXN 2000001
char s[MAXN];
int fail[MAXN];
bool search(char *str)
{
  for (int i = 0, j = 0; str[i]; i++){
    while (j && str[i] != s[j])j = fail[j - 1];
    if (str[i] == s[j] && !s[++j])return true;
  }
  return false;
}
```

定理2：KMP算法的时间复杂度为Θ(|T|)，T为文本。
证明已由前面指出。

14

第一节 单模式匹配

ACM XJTU

下面讨论 fail数组的建立算法。
通常方法是递推的建立，即由已知的fail[0]到fail[i-1]推出fail[i]。思想如下：

X	Y	X

W	a	Y	W	a

W	a	Y	W

设Border(S[0,i])=X，且X非空，则X可以写成Wa；
故W是S[0,i-1]的border。

故问题转化为求最长的border(S[0,i-1])，设为W，满足S[|W|]=S[i]。

15

第一节 单模式匹配

ACM XJTU

由定理1，我们已经能够由已知的fail[0]到fail[i-1]找到所有的border(S[0,i-1])，然后逐一判断S[|W|]=S[i]的条件是否满足即可。

算法2：fail数组的建立

```
void make_fail()
{
  for (int i = 1, j = 0; s[i]; i++){
    while (j && s[i] != s[j])j = fail[j - 1];
    if (s[i] == s[j])fail[i] = ++j;
    else fail[i] = 0;
  }
}
```

定理3：fail数组的建立时间复杂度为Θ(|S|)。

16

第一节 单模式匹配

ACM XJTU

证明：只考虑内层循环，每执行一次j至少减1。
而除内层循环外，j最多被增加|S|，这就证明了内层循环执行次数为O(|S|)。
故建立时间复杂度为Θ(|S|)。

事实上，search函数与make_fail函数是极为相似的，上述证明同样适用于KMP算法。

定理4：设模式串为S，文本为T，则KMP算法实现单模匹配的时间复杂度为Θ(|S|+|T|)。

KMP算法是一个十分高效、简洁、优美的算法！

17

第一节 单模式匹配

ACM XJTU

1.4 border、fail的进一步性质

定义6：周期与循环节
设字符串S满足存在整数k<|S|，使得对任意i∈[0,|S|-k)，有S[i]=S[i+k]，则称k为S的一个周期。
若k为S的一个周期且k | |S|，则称S严格以k为周期。S的最小严格周期中将S[0,k-1]称为S的循环节。

例2：字符串abaabaabaa以9、6、3为周期。
字符串aaaa以1、2、3为周期，以1、2为严格周期。

18

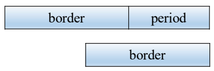
第一节 单模式匹配

ACM XJTU

定理5：S以k为周期当且仅当S[0,|S|-k-1]是S的border。

证明： 设S以k为周期，则S[i]=S[i+k]，故S[0,|S|-k-1]是S的border；

设S[0,|S|-k-1]是S的border，则对i∈ [0,|S|-k-1]，有S[i]=S[i+k]。



定理6：S以k为严格周期当且仅当S[0,|S|-k-1]是S的border且k | |S|。

定理7：若|S|-fail[|S|-1] | S且fail[|S|-1]≠0，则S的最小严格周期为|S|-fail[|S|-1]。

19

第一节 单模式匹配

ACM XJTU

定理8：若p,q均是S的严格周期，则gcd(p,q)是S的严格周期。

定理9：若p,q均是S的周期，且p+q≤|S|，则gcd(p,q)是S的周期。

证明：不妨设p<q，先证明S以q-p为周期。

（1）i≥p，则S[i]=S[i-p]=S[i+q-p]

（2）i<|S|-q，则S[i]=S[i+q]=S[i+q-p]

按照欧几里得算法思想，即证明S以gcd(p,q)为周期。

本节课暂不继续深入下去，有兴趣者可进一步研究字符串周期理论。

20

第一节 单模式匹配

ACM XJTU

1.5 KMP算法的应用

编程题1：给定模式S和文本T，判断S在T中出现了多少次？出现位置可以相交。|S|,|T|≤10^6。

```
#define MAXN 2000001
char s[MAXN];
int fail[MAXN];
int search(char *str)
{
  int ans = 0;
  for (int i = 0, j = 0; str[i]; i++){
    while (j && str[i] != s[j])j = fail[j - 1];
    if (str[i] == s[j] && !s[++j])ans++;
  }
  return ans;
}
```

21

第一节 单模式匹配

ACM XJTU

编程题2：给定模式S和文本T，T中最多找到多少个S？出现位置不能相交。|S|,|T|≤10^6。

思路：采取如下贪心：T中一旦被S匹配，即选取该匹配。

```
#define MAXN 2000001
char s[MAXN];
int fail[MAXN];
int search(char *str)
{
  int ans = 0;
  for (int i = 0, j = 0; str[i]; i++){
    while (j && str[i] != s[j])j = fail[j - 1];
    if (str[i] == s[j] && !s[++j]){ans++;j=0;}
  }
  return ans;
}
```

22

第一节 单模式匹配

ACM XJTU

编程题3：实验中想要测试转子转动情况下受力F的正弦函数。现测得了t=1,2,…,n时刻的F值，试分析该数据得出转子转动的周期。n≤10^6。

样例：

10

1 3 4 3 1 -1 -2 -1 1 3

输出：

8

字符串理论不仅仅用于解决字符串！

23

第二节 Trie树

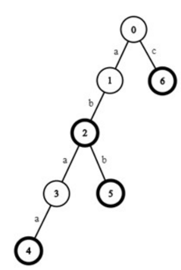
ACM XJTU

第二节 Trie树

2.1 Trie树定义与操作

定义7：Trie树是一棵树，每条边上的值为∑上的符号，且满足同一节点和孩子相连的所有边上符号均不相同。

每个由字符串组成的集合都对应一棵Trie树。



24

第二节 Trie树

ACM XJTU

Trie树的存储：

```
#define LETTER 26
struct Trie{
  int num;
  int next[LETTER];
}trie[500001];
int cnt;
```

num表示结点标记。

Trie树的初始化：

```
void init(){
  cnt = 0;
  memset(trie, 0, sizeof(Trie));
}
```

表示一棵空树。

第二节 Trie树

ACM XJTU

Trie树的插入：

```
void insert(char *s)
{
  int cur = 0;
  for (int i = 0; s[i]; i++){
    int &pos = trie[cur].next[convert(s[i])];
    if (!pos){
      pos = ++cnt;
      memset(&trie[cnt], 0, sizeof(Trie));
    }
    cur = pos;
  }
  trie[cur].num++;
}
```

时间复杂度：O(|S||∑|)

第二节 Trie树

ACM XJTU

Trie树的查找：

```
int search(char *s)
{
  int cur = 0;
  for (int i = 0; s[i]; i++){
    cur = trie[cur].next[convert(s[i])];
    if (!cur)return -1;
  }
  return trie[cur].num;
}
```

时间复杂度：O(|S|)

Trie树另一种存储方式：

`int next[LETTER];`改为`map<int> next[LETTER];`

此时插入和查询操作均为O(|S|log|∑|)

第二节 Trie树

ACM XJTU

Trie树的删除：

只需要查找到删除的字符串，将num属性减1即可。

Trie树还可以用于统计信息。例如，对每个结点记录一个total属性，可用于记录子树中有多少个字符串。

第二节 Trie树

ACM XJTU

Trie树的遍历：

```
char temp[1001];
void dfs(int i, int h)
{
  if (trie[i].num){
    temp[h] = 0;
    printf("%s %d\n", temp, trie[i].num);
  }
  for (int j = 0; j < LETTER; j++){
    if (trie[i].next[j]){
      temp[h] = convert2(j);
      dfs(trie[i].next[j], h + 1);
    }
  }
}
```

调用`dfs(0,0)`实现遍历。

第二节 Trie树

ACM XJTU

2.2 Trie树应用

应用1：实现字符串排序

设排序量为n，单位串长O(S)，总串长L，则

传统排序：时间复杂度O(nSlogn)

Trie树：时间复杂度O (|L||∑|)

应用2：取代字符串集合

Trie树插入复杂度O(|S||∑|)，集合插入复杂度O(|S|logn)

Trie树查询复杂度O(|S|)，集合查询复杂度O(|S|logn)

总之，当字母表不大时，Trie树十分高效。

应用3：二进制上的应用

编程题4：给定n个数，试选出两个，使其异或值最大。2≤n ≤10^6，每个数不超过10^9。

思路：每个数看做一个01串，将这些串建立Trie树，很容易想出如何在Trie树中寻找异或值最大、最小的两个值。
时间复杂度O(nlog10^9)。

第三节 多模式匹配

多模式匹配是给出多个模式串，查询文本中是否存在每个模式串。

考虑KMP算法，设模式串有n个，总长度为L，文本为T，则KMP算法时间复杂度为Θ(L+n|T|)。

显然通常|T|远大于L，算法十分低效。

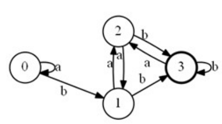
是否存在一个算法能把n去掉呢？

答案：把KMP和Trie树相结合的AC自动机！

3.1 自动机简介

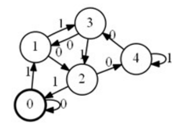
定义8：自动机
自动机是一个5元组(D,∑,δ,S,T)，其中D称作状态集合，∑ 为符号表，S∈D称为开始状态，T⊆D称为结束状态集合，δ为转移函数，定义如下：
对每个状态Q ∈D和符号a ∈ ∑，δ(Q,a)=P表示状态Q沿字符a到达状态P。

自动机实际上是一张有向图。以后默认结点0为开始状态，边缘加粗的结点为结束状态。

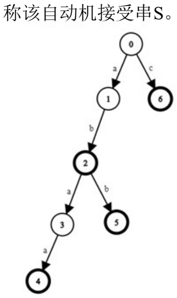


对于给定的串S，从开始状态出发，沿边上对应的字符前进，若最终能到达结束状态，称该自动机接受串S。

例3：以下自动机实现了判断一个2进制整数是不是5的倍数。



例4：以下自动机实现了判断字符串是不是abaa、abb、ab、c中的一个。

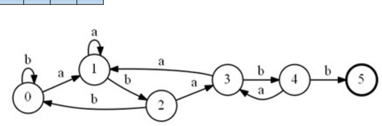


3.2 AC自动机引入

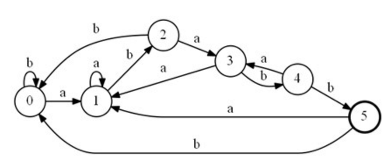
先考虑单模式匹配：利用KMP的思想，自动机状态i表示已经匹配了前i个字符，最后一个状态是结束状态。下面只要想办法构造转移函数即可。以下假设字母表∑={a,b}。

i	fail[i]
0	0
1	0
2	1
3	2
4	0

a	b	a	b	b

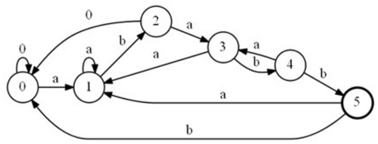


考虑到实际应用中，往往要求找到文本中的所有模式串，故结束状态处同样也加上转移。



这就是一个AC自动机！
显然，给定文本S，AC自动机的匹配时间复杂度为O(|S|)。

可以看出，对于单模式而言，给定fail数组，构造自动机的转移函数方法如下：
对于状态i，符号a，若s[i+1]=a，显然δ(i,a)=i+1；
否则，j从i开始，不断令j=fail[j]-1，直到s[j+1]=a，此时δ(i,a)=j+1；
若最后j=-1，则δ(i,a)=0。



第三节 多模式匹配 ACM XJTU

3.3 AC自动机的构造

下面考虑多模式串下AC自动机的构造。以模式串abaa,aa,ba为例。

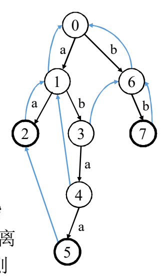
为了使用之前单模式串的思想，需要将单模式串中的fail数组拓展到多模式串。

为了将多个模式串统一成一个整体，并方便后续操作，先构造一棵对应的Trie树。之后的AC自动机均是以Trie树为基础构造的。

Trie树的结点是AC自动机的状态，Trie树的边对应了一部分转移函数。开始状态、结束状态也都确定了。

第三节 多模式匹配 ACM XJTU

引入fail指针：和KMP中的fail数组完全类似，指向匹配尽可能多字符的状态，如右图所示。

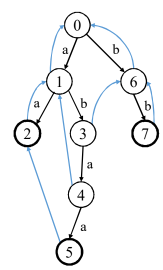


两个关键问题：
（1）如何构造fail指针？
（2）如何根据fail指针构造自动机？

fail数组是一个递推构造的过程，fail指针同样如此。注意到fail指针始终指向离Trie树树根更近的位置，因而fail指针则需要按照状态离根的距离来递推构造。

第三节 多模式匹配 ACM XJTU

初始化：Trie树根结点的fail指针为-1。
Trie树第一层的结点，fail指针指向根结点。
其他结点：设结点i的父亲为j，j到i边的符号为a。
重复j=trie[j].fail，直到结点j有符号a出发的边，则
trie[i].fail=trie[j].next[a];
若j已经为-1，则结点i的fail指针指向根结点。

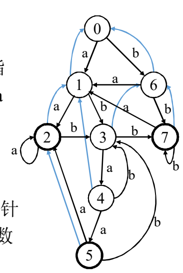


该算法是沿BFS顺序进行的。

第三节 多模式匹配 ACM XJTU

根据fail指针构造自动机：

为了求δ(Q,a)，从Q开始不断沿fail指针开始走，直到走到的状态有符号a出发的边，设该边指向的结点为P，则δ(Q,a)=P。



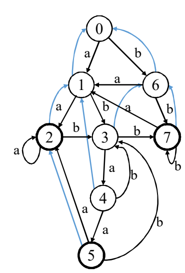
定理10：采用上述思想，构造fail指针的时间复杂度为Θ(L)，构造转移函数的时间复杂度为Θ(L|∑|)，其中L为Trie树中结点个数。

第三节 多模式匹配 ACM XJTU

3.4 AC自动机的匹配

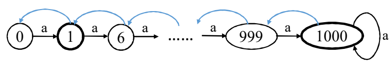
例5：在自动机上运行文本abaababbb。

可以发现，到达状态5后，不仅匹配了模式abaa，也匹配了模式aa。一般地，每到达一个状态，即使该状态不是结束状态，也要沿fail指针不断前进，检查是否和更短的模式串匹配。



事实上，按刚才的做法，每次到达一个状态，都沿fail指针向上找匹配，这样的复杂度可能非常大。这和之前求fail指针是不同的。下面是一个例子：

例6：模式串为a和aaaaaaa……（设有1000个a）共2个，文本为aaaaaaa……（设有10^6个a），自动机如下所示：



可以发现，到达状态1000后，每次都要沿fail指针向前999次，才能找到匹配的模式串a，时间复杂度相当高。

解决办法：每个结点增加一个属性，表示该结点沿fail指针到达的最近一个结束状态，记为match。

递推求match非常容易：

```
trie[i].match = trie[i].num ? i :
trie[trie[i].fail].match;
```

求match属性的时间复杂度为Θ(L)，其中L为Trie树中结点个数。

增加属性match后，每到达一个状态，沿match属性前进，每次必然匹配。

3.5 AC自动机的实现

结构体定义与初始化：

```
#define LETTER 26
struct Trie{
    int num, fail;
    int next[LETTER];
}pool[500001];
Trie* const trie = pool + 1;
int cnt;
void init(){
    cnt = 0;
    memset(pool, 0, 2 * sizeof(Trie));
    trie[0].fail = -1;
}
```

这里使用了一点技巧。

实际情况中，fail指针和状态转移函数构造是同时进行的，均按照BFS顺序。

这样做的好处：

（1）求fail指针简化为一句话：设结点i有一条沿符号a到结点j的边，则j的fail指针为

```
trie[trie[i].fail].next[a];
```

（2）求转移函数简化为一句话：设Trie树中j的父亲是i，则状态j沿字符b的转移为

```
trie[trie[i].fail].next[b];
```

值得注意的是，这是一个常数优化，没有改变时间复杂度。

算法3：AC自动机构造

```
void build()
{
  queue<int> q; q.push(0);
  while (!q.empty()){
    int t = q.front(); q.pop();
    for (int i = 0; i < LETTER; i++){
      int &cur = trie[t].next[i];
      if (cur){
        q.push(cur);
        trie[cur].fail = trie[trie[t].fail].next[i];
        trie[cur].match = trie[cur].num ? cur :
          trie[trie[cur].fail].match;
      }
      else cur = trie[trie[t].fail].next[i];
    }
  }
}
```


定理11：AC自动机的构造复杂度为Θ(L |∑|)。

算法4：AC自动机匹配

```
int search(char *s)
{
    int ret = 0, cur = 0;
    for (int i = 0; s[i]; i++){
        cur = trie[cur].next[convert(s[i])];
        for (int temp = trie[cur].match; temp;
            temp = trie[trie[temp].fail].match)
            ret += trie[temp].num;
    }
    return ret;
}
```

定理12：AC自动机匹配算法时间复杂度为Θ(|S|+m)，其中m为匹配成功次数。

3.6 AC自动机的基本应用

编程题5：给定一些模式串，问文本T被这些模式串匹配共多少次。|T|≤10^6，模式串各不相同，且长度和L不超过10^6。

思路：考虑极端情况，模式串为a,aa,aaa,……，文本为aaa……，此时匹配次数可能达到10^6×1000，若一个一个数，肯定会超时。此时需要预处理出每个状态能匹配多少次。时间复杂度为Θ(L |∑|+|T|)。

更多应用在后面介绍。

第四节 回文串基础

4.1 回文串定义

定义9：回文串
设S=S[0],S[1],…,S[|S|-1],
定义Rev(S)= S[|S|-1],S=[|S|-2],…,S[0],
若S=Rev(S)，称S是回文串。

例7：a，aba，abba，aaaa都是回文串。

4.2 Manacher算法思想

Manacher算法用于求解字符串S以每一位置为中心（包括两字符中间为中心）的最长回文子串。

朴素算法：对每一位置为中心分别向两侧延伸，找到这一位置的最长回文子串。

时间复杂度O(|S|^2)。例如对于串aaaaaaaaaa。

低效的原因：完全没有考虑这些不同中心的回文串之间的关系。 Manacher算法则利用了回文串之间的联系。

预处理：将字符串每两个字符中间插入一无用字符（如'#'），前后各插入另一无用字符（如'*'）。

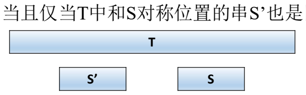
a		b		a		a		b
1		3		1		1		1
	0		0		4		0	

*	#	a	#	b	#	a	#	a	#	b	#	*
		1		5		3		3		1		
			1		1		9		1			

预处理后，所有位置的答案均乘2加1。

预处理目的：避免边界；相邻字符不相同；所有回文子串都以字符为中心。

Manacher算法原理：若S是T的子串，T是回文串，则S是回文串当且仅当T中和S对称位置的串S'也是回文串。



仅用到这一原理，就产生了Manacher算法。该算法仍然是从左往右枚举回文中心，但是如果该中心位置在某一回文子串内部，那么就直接看对称位置的答案即可。

*	#	a	#	b	#	b	#	a	#	b	#	*
	1	3	1	3	9	3	1	7	1	3	1	

可以看出：Manacher算法执行流程如下：
从左往右枚举回文中心。对当前回文中心，首先看该中心是否在矩形框所示的回文子串内。
（1）若不在，则暴力求出该回文中心对应的最长回文子串，同时矩形框更新为该子串；
（2）若在，查询对称位置的答案（设为t），此时有两种情况：
a.对称位置的回文串没有触碰或超过矩形框左边界，则答案为t；
b.对称位置的回文串触碰或超过了矩形框左边界，则从右边界开始暴力求出答案，同时矩形框更新为该子串。

第四节 回文串基础

ACM XJTU

4.3 Manacher算法实现

算法5：Manacher算法

```
char s[1000001], s2[2000001];
int ans[2000002];
int manacher()
{
    int i = 0, pos = 0, j = 0, ret = 0;
    s2[0] = '$';
    do{
        s2[2 * i + 1] = '#';
        s2[2 * i + 2] = s[i];
    } while (s[i++]);
    ans[0] = 0;
```


第四节 回文串基础

ACM XJTU

```
    for (i = 1; s2[i]; i++){
        int t = i >= j ? 0 :
            min(j - i, ans[pos * 2 - i]);
        while (s2[i + t + 1] == s2[i - t - 1])t++;
        if (i + t>j){ j = i + t; pos = i; }
        ans[i] = t;
        ret = max(ret, t);
    }
    return ret;
}
```

定理13：对字符串S执行Manacher算法的时间复杂度为Θ(|S|)。

注意到每次操作，图示中蓝色指针和红色指针必有一个向右移，而总的移动次数是Θ(|S|)的。

第五节 应用

ACM XJTU

第五节 应用

5.1 KMP与串的去重连接

编程题6：字符串连接有时是需要首尾去重的。例如，将串“(1+2)*3=3*3”和串“3*3=9”相连接，我们希望得到“(1+2)*3=3*3=9”。现在给定串S和T，请进行尽可能长的首尾去重，并输出去重连接结果。|S|,|T|≤10^6。

思路：用串T匹配S，找到首次匹配到S末尾时的位置，这段区间即为最大去重长度。采用KMP算法，时间复杂度Θ(|S|+|T|)。

第五节 应用

ACM XJTU

5.2 AC自动机与DP

编程题7：DNA改造

众所周知，DNA上的某些片段将导致遗传性疾病。现在给出一些有疾病的DNA片段（总长度L不超过1000），并给出一条DNA链S（长度不超过10000）。S上可能含有疾病的片段。问最少改变S上的几个碱基对，可以使其不包含有疾病的DNA。注意，DNA链上的碱基对只有A,G,C,T四种。例如有疾病的DNA片段为“A”和“ TG ”，S=TGAATG，则答案为4。

第五节 应用

ACM XJTU

对所有疾病DNA建立AC自动机。

以状态dp[i][j]表示前i个字符到达状态j的情况下最少修改几个字符，则转移方程为:

```
dp[i + 1][cur] = min(dp[i + 1][cur], dp[i][j] +
(convert(s[i]) != k));
```

其中k为读入的下一个字符，

```
cur = trie[j].next[k];
```

注意：`cur`状态不能匹配任意疾病字符串，这可由`match`属性判断。

初始条件：`dp[0][0]=0;`其他为无穷

时间复杂度：O(L|S||∑|)。

第五节 应用

ACM XJTU

5.3 AC自动机与矩阵快速幂

编程题8：一个程序由01代码组成，若这个01串中含有某种序列将会被360识别为病毒。请问长度为n的程序中不会被360识别为病毒的程序有几个？360的病毒库大小只有不到100个01字符，n≤10^9。答案对10^9+7取模。

定理14：设图G的邻接矩阵为A，则从s经过k条边到t的路径数为矩阵A^k第s行第t列的元素值。

思路：将AC自动机看做有向图，去掉所有结束状态后，求0状态经过k条边到所有状态的路径数之和。时间复杂度O(L^3×logn)。

第五节 应用

ACM XJTU

5.4 AC自动机的其他应用

与图论中强连通分量结合，解决是否存在无限长的文本，不包含任意指定模式串。

与线性方程组求解相结合，解决概率问题。
有n个人，每人说一段1到6构成的序列，然后开始掷骰子，最先出现自己说的序列的人获胜。求每人获胜的概率。

时间关系，不展开介绍了。

61

进阶内容

ACM XJTU

进阶内容：

1、后缀数组

2、后缀自动机

玄学内容：字符串Hash

难点：

自动机与图论、数据结构的结合应用

62

ACM XJTU

谢谢！

63