

ACM Template

Zeng Xiaocan

August 13, 2019

Contents

1	String	3
1.1	STL	3
1.2	Max/Min-Expression	3
1.3	KMP	4
1.4	EXKMP	5
1.5	Hash	6
1.6	Trie	6
1.7	AC-Automaton	7
1.8	Manacher	8
1.9	Palindromic-Tree	9
1.10	Suffix-Array	11
1.11	Suffix-Automaton	11
1.12	ProblemSet	12

1 String

1.1 STL

```
reverse(s.begin(), s.end());
transform(s.begin(), s.end(), s.begin(), ::toupper); (::tolower)
//字符串和数字互转
int a;
stringstream(s) » a;
char s[100];
sprintf(s,"%d",a);
string(v.begin(),v.end());
//返回 pos 开始的长度为 len 的字符串
substr(pos,len);
//在 pos 位置插入字符串 s
insert(int pos,string s)
//从索引 pos 开始往后删 num 个，num 为空表示全删除
erase(pos,num);
//删除迭代器 it 指向的字符，返回删除后迭代器的位置
erase(it);
//删除迭代器 [first, last) 之间的所有字符，返回删除后迭代器的位置
erase(first,last);
//从 pos 开始查找字符 c/字符串 s 在当前字符串的位置
int find(c/s,pos);
```

1.2 Max/Min-Expression

```
//求循环字符串 s 的最小/最大表示
//i, j: 当前比较两个字符串的起始位置
//k: 这两个字符串已比较的长度
int getMin(char s[]){
    int n=strlen(s);
    int i=0,j=1,k=0;
    while(i<n && j<n && k<n){
        int t=s[(i+k)%n]-s[(j+k)%n];
        if(!t){
            k++;
        }else{
            if(t>0){
                //如果是求最大表示则为 j+=k+1
                i+=k+1;
            }else{
                j+=k+1;
            }
        }
        if(i==j){
            j++;
        }
    }
}
```

```
        }
        k=0;
    }
}
return min(i,j);
}
```

1.3 KMP

//nex[i]: 表示前 i 个字符的最长相同前后缀长度

```
void getNext(char s[],int n){
    int i=0,j=-1;
    nex[0]=-1;
    while(i<n){
        if(j==-1 || s[i]==s[j]){
            nex[++i]=++j;
        }else{
            j=nex[j];
        }
    }
}
```

//前 i 个字符的最小循环节长度: $i - nex[i]$, 个数: $i / (i - nex[i])$

```
int kmp(char s[],int n,char p[],int m){
    int i=0,j=0;
    // int cnt=0;
    getNext(p,m);
    while(i<n && j<m){
        if(j==-1 || s[i]==p[j]){
            i++;
            j++;
        }else{
            j=nex[j];
        }
        if(j==m){
            //匹配位置
            return i-j+1;
            //匹配个数
            //cnt++;
            //不可重叠
            //j=0;
            //可重叠
            //j=nex[j];
        }
    }
    //return cnt;
}
```

1.4 EXKMP

//nex[i] 表示 t 串中以 i 开始的后缀与 t 串的最长公共前缀

//ext[i] 表示 s 串中以 i 开始的后缀与 t 串的最长公共前缀

```
void getNext(char *t,int len){
    int a=0;
    while(a<len-1 && t[a]==t[a+1]){
        a++;
    }
    nex[1]=a;
    int po=1;
    for(int i=2;i<len;i++){
        int p=po+nex[po]-1;
        int l=nex[i-po];

        if(l>=p-i+1){
            int j=max(0,p-i+1);
            while(i+j<len && t[i+j]==t[j]){
                j++;
            }
            nex[i]=j;
            po=i;
        }else{
            nex[i]=1;
        }
    }
}

void getExt(char *s,int n,char *t,int m){
    int a=0;
    getNext(t,m);
    int mlen=min(n,m);
    //计算 ext[0]
    while(a<mlen && s[a]==t[a]){
        a++;
    }
    ext[0]=a;
    //po 表示当前最右的 i+ext[i]-1 所对应的 i
    int po=0;
    for(int i=1;i<n;i++){
        //p 表示最右的 i+ext[i]-1
        int p=po+ext[po]-1;
        //此时前面已匹配的 s[po..p]==t[0..p-po], 即 s[i..p]==t[i-po..p-po]
        //所以 l 就是表示 t[i-po...m-1] 和 t[0..m-1] 的 lcp
        //也就是 s[i..p] 和 t[0..m-1] 的 ** 部分 **lcp
        //得看 l 和 p-i+1(ext[i] 可能的最大值) 哪个大
        int l=nex[i-po];
        if(l>=p-i+1){
```

```

        //l 大, 那么从 p-i+1(目前可以保证的 ext[i] 的值) 继续暴力往下匹配
        int j=max(0,p-i+1);
        while(i+j<n && j<m && s[i+j]==t[j]){
            j++;
        }
        ext[i]=j;
        po=i;
    }else{
        //p-i+1 大, 那么 ext[i] 就只能是 1 了
        ext[i]=1;
    }
}
}
}

```

1.5 Hash

//单哈希很容易卡；取模很慢

```

ull seeds[]={27,146527,19260817,91815541};
ull mods[]={1000000009,998244353,4294967291ull,21237044013013795711};
struct Hash{
    ull seed,mod;
    ull bas[N];
    ull sum[N];
    void init(int sidx,int midx,int len,char *s){
        seed=seeds[sidx];
        mod=mods[midx];
        bas[0]=1;
        for(int i=1;i<=len;i++){
            bas[i]=bas[i-1]*seed%mod;
        }
        for(int i=1;i<=len;i++){
            sum[i]=(sum[i-1]*seed%mod+s[i])%mod;
        }
    }
    ull getHash(int l,int r){
        return (sum[r]-sum[l-1]*bas[r-l+1]%mod+mod)%mod;
    }
}hs;

```

1.6 Trie

//val[u] 表示 u 节点处保存的单词数

```

struct Trie{
    int cnt,tr[N][26],val[N];
    void insert(char *s){
        int len=strlen(s);
        int now=0;
    }
}

```

```

        for(int i=0;i<len;i++){
            int id=s[i]-'a';
            if(!tr[now][id]){
                tr[now][id]=++cnt;
            }
            now=tr[now][id];
        }
        val[now]++;
    }
}T;

```

1.7 AC-Automaton

//*fail*[*x*] 指向以 *x* 为结尾的后缀在 ** 其他模式串中 ** 所能匹配的最长前缀
 //当 *tr*[*now*][*i*] 失配时, 就可以跳转到以已匹配的这部分后缀作为前缀的其他模式串。

```

struct ACM{
    int tr[N][26],val[N],fail[N],cnt;
    void insert(char *s){
        int len=strlen(s);
        int now=0;
        for(int i=0;i<len;i++){
            int id=s[i]-'a';
            if(!tr[now][id]){
                tr[now][id]=++cnt;
            }
            now=tr[now][id];
        }
        val[now]++;
    }
    //比 Trie 树多了构建 fail 指针
    void build(){
        queue<int> q;
        //初始化第一层
        for(int i=0;i<26;i++){
            if(tr[0][i]){
                fail[tr[0][i]]=0;
                q.push(tr[0][i]);
            }
        }
        while(!q.empty()){
            int u=q.front();
            q.pop();
            for(int i=0;i<26;i++){
                if(tr[u][i]){
                    fail[tr[u][i]]=tr[fail[u]][i];
                    q.push(tr[u][i]);
                }
            }
        }
    }
}

```

```

        }else{
            tr[u][i]=tr[fail[u]][i];
        }
    }
}
}
//查询所有模式串出现的总次数
int query(char *s){
    int len=strlen(s);
    int ans=0;
    int now=0;
    for(int i=0;i<len;i++){
        int id=s[i]-'a';
        now=tr[now][id];
        //打标记暴力跳 fail, 避免重复计数
        for(int t=now;t && val[t]!=-1; t=fail[t]){
            ans+=val[t];
            val[t]=-1;
        }
    }
    return ans;
}
}ac;

```

1.8 Manacher

//ma[]: 新字符串 (ma, mp 都注意要开两倍空间!)
 //mp[i]: 表示以 i 为中心的回文子串的半径 (包括特殊字符)
 //mx: 能延伸到最右端的位置
 //id: 能延伸到最右端的回文串中心位置

```

void manacher(char s[],int len){
    //构造新字符串, 两个字符之间插入一个其他字符, 第 0 个字符忽略 (即加入另一种字符)
    int l=0;
    ma[l++]='$';
    ma[l++]='#';
    for(int i=0;i<len;i++){
        ma[l++]=s[i];
        ma[l++]='#';
    }
    ma[l]='\0';
    int mx=0,id=0;
    for(int i=1;i<l;i++){
        //若 mx>i: mp[2*id-i] 表示 i 关于 id 的对称点的最长回文半径
        //不能超出 mx, 所以和 mx-i 取 min
        //若 mx<i: mp[i]=1
        mp[i]=mx>i?min(mp[2*id-i],mx-i):1;
    }
}

```



```

//往两边更新
while(ma[i+mp[i]]==ma[i-mp[i]]){
    mp[i]++;
}
//更新全局 mx 和 id
if(i+mp[i]>mx){
    mx=i+mp[i];
    id=i;
}
}
}

```

1.9 Palindromic-Tree

```

struct PT{
    //回文树中每个节点表示一个回文串，所以有偶数长度的树和奇数长度的树两棵
    //next 指针 next[u][i] 表示 u 节点左右添加字符 i 之后得到的回文串节点
    int next[N][26];
    //fail 指针 失配后跳转到最长后缀回文串对应的节点
    int fail[N];
    //节点对应回文串在原串中出现次数，需先调用 count 函数
    int cnt[N];
    //num[i] 表示 ** 以节点 i 所表示的回文串右端点结尾 ** 的回文串个数（包括自身）
    //即 fail 指针的深度
    int num[N];
    //节点对应回文串的长度
    int len[N];
    //存放添加的字符
    int S[N];
    //上一个字符所在节点
    int last;
    //节点对应的最新字符位置，反向映射 last(可以改成 vector<int>[])
    int id[N];
    //字符数，不等于节点数
    int n;
    //回文树总结点数，包括奇偶两个空节点，节点编号为 0 到 p-1
    //不同回文子串个数 p-2 回文子串个数 \sum num[i]
    int p;
    //创建长度为 l 的新节点
    int newnode(int l){
        for(int i=0;i<26;i++){
            next[p][i]=0;
        }
        cnt[p]=0;
        num[p]=0;
        len[p]=1;
    }
}

```

```

        return p++;
    }
    //初始化
    void init(){
        p=0;
        //奇偶空节点，先偶再奇
        newnode(0);
        newnode(-1);
        last=0;
        n=0;
        S[n]=-1;
        //偶根 fail 指向奇根
        fail[0]=1;
    }
    //找到新插入字符 c 的回文匹配位置
    int getFail(int x){
        //在节点 x 对应串的后面加上一个字符，就判断 x 前面字符是否相同
        //若相同直接构成新的回文串，不同就跳到 fail，即最长回文后缀
        //S[n-len[x]-1] 就是新加的字符 (S[n]) 关于 x 串的对称字符
        while(S[n-len[x]-1]!=S[n]){
            x=fail[x];
        }
        return x;
    }
    //插入字符 c
    void add(int c){
        c-='a';
        S[++n]=c;
        //找到当前回文串匹配位置，也就是当前回文串节点的父节点
        int cur=getFail(last);
        if(!next[cur][c]){
            //出现了一个新的本质不同的回文串
            int now=newnode(len[cur]+2);
            //类似于 AC 自动机，往上跳直到找到满足条件的串节点
            //getFail 其实就是不断比较当前加入的字符和 x 节点对称的那个字符
            fail[now]=next[getFail(fail[cur])][c];
            //fail 指针深度加 1
            num[now]=num[fail[now]]+1;
            //这句要放最后，前面的指针关系处理好再连上子节点
            next[cur][c]=now;
        }
        //最新回文串节点
        last=next[cur][c];
        cnt[last]++;
        id[last]=n;
    }
}

```

```

//统计每个节点回文串出现次数
void count(){
    //从子节点逆推
    for(int i=p-1;i>=0;i--){
        //i 节点出现, 说明其最长回文后缀 fail[i] 也出现
        cnt[fail[i]]+=cnt[i];
    }
}
}ac;

```

1.10 Suffix-Array

1.11 Suffix-Automaton

//空间足够的情况下开大点

```

struct SAM{
    //转移边
    int next[N*2][26];
    //link 边
    int fa[N*2];
    //状态内最长后缀长度
    int len[N*2];
    //状态对应 endpos 大小, 即子串出现次数
    int num[N*2];
    //总节点数
    int cnt;
    //上一个节点
    int lst;
    int newnode(int l,int s){
        for(int i=0;i<26;i++){
            next[cnt][i]=0;
        }
        len[cnt]=l;
        num[cnt]=s;
        return cnt++;
    }
    //初始化
    void init(){
        cnt=0;
        lst=newnode(0,0);
        fa[lst]=-1;
    }
    void add(int c){
        c-='A';
        int p=lst;
        int cur=newnode(len[p]+1,1);
        //假设当前 sam 为 "aabb", 起点 S 为空串, 节点 5 是 {b}, 节点 4 是 {aabb,abb,bb}

```

```

//定义 suffix-path 为当前字符串的所有后缀的状态, 即  $S[1..i]$ ,  $S[2..i]...$ 
//此时的 s-p 就是  $S-5-4$ , (b 这个后缀因为 endpos 大于其他, 所以在节点 5)
//插入的字符是 a, 而 s-p 上 5 和 4 节点都没有 a, 因此将节点 5 和 4 fa 节点 6
//节点 6 此时为  $\{aabba, abba, bba, ba\}$ 
//当路径上的节点已经有 a
while(p!=-1 && !next[p][c]){
    next[p][c]=cur;
    p=fa[p];
}
if(p==-1){
    //对应上面整个路径都没有 a 的情况
    fa[cur]=0;
}else{
    //路径上找到一个有 a, 往前肯定都有 a
    int q=next[p][c];
    if(len[q]==len[p]+1){
        //这里节点  $S(p)$  为空串, 而节点  $1(q)$  为  $\{a\}$ , 因此将新节点 6 fa 节点 1
        fa[cur]=q;
    }else{
        //st[q].len>st[p].len+1
        //假设当前 sam 为 "aab", 起点 S 为空串, 节点 4 是  $\{aab, ab, b\}$ 
        //此时的 s-p 就是  $S-3$ , 要插入的字符是 b, 路径上 S 节点有 b, 指向节点 3
        //而  $st[3].len>st[S].len+1$ , 因此需要将节点 3 拆分
        //把从节点  $S+b$  得到的后缀  $\{b\}$  分给新的节点 5
        //将 q 拆成两个节点,  $p \rightarrow cl \rightarrow new$ 
        int cl=newnode(len[p]+1,0);
        fa[cl]=fa[q];
        memcpy(next[cl],next[q],sizeof(next[cl]));
        while(p!=-1 && next[p][c]==q){
            //之前路径上所有 p 走向 q 的, 现在全部走向 q 拆出的新节点
            next[p][c]=cl;
            p=fa[p];
        }
        //q 和新节点都 fa 向拆出节点
        fa[q]=fa[cur]=cl;
    }
}
//更新最后一个节点
lst=cur;
}
}ac;

```

1.12 ProblemSet