子串与后缀

目录

后缀自动机(Suffix Automaton, SAM)是处理字符串问题非常强大的工具。

几乎所有涉及到字符串子串的问题，都可以用后缀自动机解决。此外，后缀自动机还可以拓展到 Trie 上，解决多字符串的问题（广义后缀自动机）。

SAM 的点数为最大不超过 $2|S|$ ，转移数不超过 $3|S|$。GSAM 点数不超过 $2|\text{Trie}|$。

所以都要预留**2倍空间**。

后缀自动机

时空复杂度 $\mathcal{O}(|S||\Sigma|)$。为了方便，之后的复杂度分析均忽略建 SAM 的时间。

如果字符集很大，将`int ch[A]`改成`std::map`即可，复杂度变为 $\mathcal{O}(|S|\log|\Sigma|)$，但是后续的在 SAM 上的操作基本会多一个 $\log$。

对长度为 10^6 的串建立 SAM 需要 `215MiB` 左右的内存（节点仅维护`link,len,ch`的情况），每多一个`int`增加`7.6MiB`左右的内存开销。

```
struct SuffixAutomaton {
    constexpr static int A=26;
    constexpr static char B='a';
    struct Endpos {
        int link,len,cnt;
        int ch[A];
    };
    vector<Endpos> edp;
```

```
    int last=0;

    int new_node() {
        edp.push_back({});
        return edp.size()-1;
    }

    void extend(char x) {
        int c=x-B;
        int p=last;
        int cur=last=new_node();
        edp[cur].len=edp[p].len+1;
        for(;p!=-1&&!edp[p].ch[c];p=edp[p].link) edp[p].ch[c]=cur;
        if(p!=-1) {
            int q=edp[p].ch[c];
            if(edp[p].len+1==edp[q].len) edp[cur].link=q;
            else {
                int clone=new_node();
                edp[clone]=edp[q];
                edp[clone].len=edp[p].len+1;
                edp[cur].link=edp[q].link=clone;
                for(;p!=-1&&edp[p].ch[c]==q;p=edp[p].link)
                    edp[p].ch[c]=clone;
            }
        }
    }

    int size() { return edp.size(); }
    void build(const string &s) { for(auto x:s) extend(x); }
    void clear() { edp.clear(),edp.push_back({-1}),last=0; }

    SuffixAutomaton(int sz=0) { edp.reserve(sz),clear(); }
    SuffixAutomaton(const string &s) {
edp.reserve(s.size()*2),clear(),build(s); }
} sam;
```

广义后缀自动机

在线构造

时间复杂度 $\mathcal{O}(G(T)|\Sigma|)$，$G(T)$ 为 Trie 叶节点深度和，也就是和所有串长度总和成正比。

由于需要存储 Trie 内存开销翻倍。

```
struct GeneralSuffixAutomaton {
    constexpr static int A=26;
    constexpr static char B='a';
    using Arr=array<int, A>;
    struct Endpos {
        int link,len;
        Arr ch;
```

```
    };
    vector<Arr> tr;
    vector<Endpos> edp;

    int new_tr() { tr.push_back({}); return tr.size()-1; }
    int new_edp() { edp.push_back({}); return edp.size()-1; }

    int split(int p,int c,int len) {
        int q=edp[p].ch[c];
        if(edp[q].len==len) return q;
        else {
            int clone=new_edp();
            edp[clone]=edp[q];
            edp[clone].len=len;
            edp[q].link=clone;
            for(;p!=-1&&edp[p].ch[c]==q;p=edp[p].link)
                edp[p].ch[c]=clone;
            return clone;
        }
    }

    void extend(int &p,int &t,char x,int len) {
        int c=x-B;
        int last;
        if(tr[t][c]) last=edp[p].ch[c];
        else {
            tr[t][c]=new_tr();
            if(edp[p].ch[c]) last=split(p, c, len);
            else {
                int cur=last=new_edp();
                edp[cur].len=len;
                for(;p!=-1&&!edp[p].ch[c];p=edp[p].link)
                    edp[p].ch[c]=cur;
                if(p!=-1) edp[cur].link=split(p, c, edp[p].len+1);
            }
        }
        t=tr[t][c];
        p=last;
    }

    void insert(string &s) {
        for(int p=0,t=0,i=0;i<s.size();i++) extend(p, t, s[i], i+1);
    }

    int size() { return edp.size(); }

    void clear() {
        edp.clear(),edp.push_back({-1});
        tr.clear(),tr.push_back({});
    }

    GeneralSuffixAutomaton(int sz=0) {
edp.reserve(sz),tr.reserve(sz),clear(); }
} sam;
```

```
```

离线构造

时间复杂度 $\mathcal{O}(|T||\Sigma|)$，由于压缩了 Trie 比在线版本节省一半内存，基本与 SAM 保持一致。

```cpp
struct GeneralSuffixAutomaton {
    constexpr static int A=26;
    constexpr static char B='a';
    using Arr=array<int, A>;
    struct Endpos {
        int link,len;
        Arr ch;
    };
    vector<Endpos> edp;

    int new_edp() { edp.push_back({}); return edp.size()-1; }

    int split(int p,int c,int len) {
        int q=edp[p].ch[c];
        if(edp[q].len==len) return q;
        else {
            int clone=new_edp();
            edp[clone]=edp[q];
            edp[clone].len=len;
            edp[q].link=clone;
            for(;p!=-1&&edp[p].ch[c]==q;p=edp[p].link)
                edp[p].ch[c]=clone;
            return clone;
        }
    }

    void extend(int p,int c) {
        int cur=edp[p].ch[c];
        edp[cur].len=edp[p].len+1;
        for(;p!=-1&&(edp[p].ch[c]==cur||!edp[p].ch[c]);p=edp[p].link)
            edp[p].ch[c]=cur;
        if(p!=-1) edp[cur].link=split(p, c, edp[p].len+1);
    }

    void insert(string &s) {
        int t=0,c=0;
        for(auto x:s) {
            c=x-B;
            if(!edp[t].ch[c]) edp[t].ch[c]=new_edp();
            t=edp[t].ch[c];
        }
    }

    void build() {
        queue<int> q;
        q.push(0);
        while(q.size()) {
```

```
                int p=q.front();
                q.pop();
                for(int c=0;c<A;c++) if(edp[p].ch[c])
                    extend(p, c),q.push(edp[p].ch[c]);
            }
        }

        int size() { return edp.size(); }
        void clear() { edp.clear(),edp.push_back({-1}); }

        GeneralSuffixAutomaton(int sz=0) { edp.reserve(sz),clear(); }
    } sam;
```

后缀数组

使用倍增法将一个串所有的后缀排序。

时间复杂度 $\mathcal{O}(|S|\log|S|)$。

```
constexpr int N=1e6+10;

int n,m=1<<7;
string s;
int fir[N],sec[N],cnt[N];
int sa[N],rk[N],height[N];

void get_sa() {
    for(int i=1;i<=n;i++) cnt[fir[i]=s[i]]++;
    for(int i=2;i<=m;i++) cnt[i]+=cnt[i-1];
    for(int i=n;i;i--) sa[cnt[fir[i]]--]=i;

    for(int k=1;k<=n;k<<=1) {
        int num=0;
        for(int i=n-k+1;i<=n;i++) sec[++num]=i;
        for(int i=1;i<=n;i++) if(sa[i]>k) sec[++num]=sa[i]-k;
        for(int i=1;i<=m;i++) cnt[i]=0;
        for(int i=1;i<=n;i++) cnt[fir[i]]++;
        for(int i=2;i<=m;i++) cnt[i]+=cnt[i-1];
        for(int i=n;i;i--) sa[cnt[fir[sec[i]]]--]=sec[i],sec[i]=0;
        swap(fir,sec);
        fir[sa[1]]=num=1;
        for(int i=2;i<=n;i++)
            fir[sa[i]]=(sec[sa[i]]==sec[sa[i-1]]&&sec[sa[i]+k]==sec[sa[i-
1]+k])
                ?num:++num;
        if(num==n) break;
        m=num;
    }
}

void get_height() {
```

```
    for(int i=1;i<=n;i++) rk[sa[i]]=i;
    for(int i=1,k=0;i<=n;i++) {
        if(rk[i]==1) continue;
        if(k) k--;
        int j=sa[rk[i]-1];
        while(i+k<=n&&j+k<=n&&s[i+k]==s[j+k]) k++;
        height[rk[i]]=k;
    }
}
```

应用

拓扑序

利用后缀链接 $link$ 进行拓扑排序，得到 $parent$ 树的拓扑序。同时根据SAM的性质可知这个序列同时也是DAG的（逆）拓扑序。初始状态不在内，如果需要用到，直接 `push_back(0)` 即可。

```
vector<int> toporder;
void toposort() {
    auto &q=toporder;
    q.clear();
    q.reserve(size());
    vector<int> ind(size());
    for(int i=1;i<size();i++) ind[edp[i].link]++;
    for(int i=1;i<size();i++) if(!ind[i]) q.push_back(i);
    for(int i=0;i<q.size();i++) {
        int u=q[i];
        int p=edp[u].link;
        if(p&&!--ind[p]) q.push_back(p);
    }
}
```

不同子串数

求串 S 有多少个本质不同的子串。利用 SAM 的性质，每个等价类包含的子串数量为 `edp[u].len-edp[edp[u].link].len`。

时间复杂度 $\mathcal{O}(|S|)$。

```
for(int i=1;i<sam.size();i++)
    ans+=sam.edp[i].len-sam.edp[sam.edp[i].link].len;
```

这个问题可以拓展到多串：求多个串本质不同的子串，使用 GSAM 即可。

子串出现次数

原串 S 的前缀 pre_i 是 endpos 含有 i 的最长串。 在 SAM 上匹配原串 S，将途径的状态 $cnt+1$。之后利用拓扑序dp即可求出每个等价类的 endpos 大小。

时间复杂度 $\mathcal{O}(|S|)$。

```
void count(const string &s) {
    int u=0;
    for(auto x:s) {
        int c=x-B;
        u=edp[u].ch[c];
        edp[u].cnt++;
    }
    for(int u:toporder) {
        int p=edp[u].link;
        edp[p].cnt+=edp[u].cnt;
    }
}
```

最长公共子串

要计算串 S,T 的最长公共子串，首先对串 S 建 SAM 然后仿照AC自动机的做法在上面匹配 T，每次可以求出 T 的前缀与 S 的最长公共子串：

- 若下一个字符不匹配，暴力跳 $link$，将当前匹配长度 len 更新为 $edp[p].len$，直到不能跳为止或者匹配。
- 若匹配，转移状态，$len+1$。

时间复杂度 $\mathcal{O}(|S|+|T|)$。

```
int match(const string &t) {
    int u=0,len=0,ans=0;
    for(auto x:t) {
        int c=x-B;
        while(u&&!edp[u].ch[c]) u=edp[u].link,len=edp[u].len;
        if(edp[u].ch[c]) u=edp[u].ch[c],len++;
        ans=max(ans,len);
    }
    return ans;
}
```

类似的还有多串最长公共子串。做法也是类似的，首先找出最短的一个串 s （否则时间复杂度无法保证），然后对 s 以外的所有串建 SAM 并同时进行匹配。

时间复杂度 $\mathcal{O}(k|s|)$，其中 k 为串数。

```
int ans=0;
for(auto x:s[idx]) {
    int c=x-SuffixAutomaton::B;
```

```
    int res=N;
    for(int i=1;i<=n;i++) {
        auto &edp=sam[i].edp;
        int &u=uid[i];
        int &l=len[i];
        while(u&&!edp[u].ch[c]) u=edp[u].link,l=edp[u].len;
        if(edp[u].ch[c]) u=edp[u].ch[c],l++;
        res=min(res,l);
    }
    ans=max(ans,res);
}
```

子串在多少原串中出现

求有多少个原串含有指定子串。

类似与求子串出现次数，但是在一个原串中出现仅算一次。首先建立 GSAM，然后对于每个串进行树上涂色操作，即可求出在该串中出现的等价类有哪些。

```
void markup(const string &s) {
    vector<int> q;
    int u=0;
    for(auto x:s) {
        int c=x-B;
        u=edp[u].ch[c];
        q.push_back(u);
        edp[u].mark=1;
    }
    for(int i=0;i<q.size();i++) {
        int u=q[i],p=edp[u].link;
        edp[u].cnt++;
        if(p&&!edp[p].mark) q.push_back(p),edp[p].mark=1;
    }
    for(int u:q) edp[u].mark=0;
}
```

多串公共子串数：求在每个原串中都出现过的不同子串数量。 涂色完毕后检查每个等价类出现的次数（颜色数量）即可。

```
LL count(int k) {
    LL ans=0;
    for(int i=1;i<size();i++)
        if(edp[i].cnt==k)
            ans+=edp[i].len-edp[edp[i].link].len;
    return ans;
}
```

时间复杂度 $\mathcal{O}(k|T|)$。不要使用这个方法求多串最长公共子串，会被卡超时。

定位子串

询问串 S 的一个子串 $S[l,r]$ 在 SAM 上的位置。

首先求出每个前缀在 SAM 上的位置，使用 pos 表示。$S[l,r]$ 的位置必然可以通过 $pos[r]$ 跳若干次 $link$ 找到（即 $S[:r]$ 前面去点一些字符）。利用树上倍增加速这一过程，即可快速求出 $S[l,r]$ 的位置。

预处理时间复杂度 $\mathcal{O}(|S|\log|S|)$，查询复杂度 $\mathcal{O}(\log|S|)$。

最长公共前缀 LCP

求串 S 的两个子串 $S[l,r],S[L,R]$ 的最长公共前缀。

两个串的公共前缀等于反串的公共后缀。首先建立 S 的反串的 SAM，然后定位两个串的的位置，记为 u,v，最长公共后缀 LCS 所在的位置便是 $lca(u,v)$，记得特判 $u=v$ 的情况。

预处理时间复杂度 $\mathcal{O}(|S|\log|S|)$，查询复杂度 $\mathcal{O}(\log|S|)$。

上面两个问题的代码可以合并到一起（下标均从0开始）：

```
// * index start from 0
namespace lca {
    const auto &edp=sam.edp;
    constexpr int M=__lg(N*2);
    int fa[N*2][M+1],dep[N*2],pos[N];

    void get_fa(const vector<int> &q) {
        dep[0]=1;
        for(auto it=q.rbegin();it!=q.rend();it++) {
            int u=*it;
            int p=edp[u].link;
            dep[u]=dep[p]+1;
            fa[u][0]=p;
            for(int i=1;i<=M;i++) fa[u][i]=fa[fa[u][i-1]][i-1];
        }
    }

    void get_pos(const string &s) {
        int u=0;
        for(int i=0;i<s.size();i++) {
            int c=s[i]-sam.B;
            u=edp[u].ch[c];
            pos[i]=u;
        }
    }

    int find(int l,int r) {
        int u=pos[r];
        int len=r-l+1;
        for(int i=M;i>=0;i--) {
            int p=fa[u][i];
            if(edp[p].len>=len) u=p;
```

```
        }
        return u;
    }

    int lca(int u,int v) {
        if(dep[u]<dep[v]) swap(u,v);
        for(int k=M;~k;k--)
            if(dep[fa[u][k]]>=dep[v])
                u=fa[u][k];
        if(u==v) return u;
        for(int k=M;~k;k--)
            if(fa[u][k]!=fa[v][k])
                u=fa[u][k],v=fa[v][k];
        return fa[u][0];
    }

    int lcs(int l,int r,int L,int R) {
        int u=find(l,r),v=find(L,R);
        if(u==v) return min(r-l+1,R-L+1);
        int p=lca(u,v);
        return edp[p].len;
    }
}
```

区间 endpos 维护

求子串在 $S[l,r]$ 区间的 endpos。

利用树上线段树合并可以维护出每个等价类的的 endpos 位置集合。具体可以参考数据结构的部分。

时空复杂度 $\mathcal{O}(|S|\log|S|)$ 。每次合并都新建节点，多一倍节点数量。设自动机节点数为 N，则节点数上界为 $2N\log|S|$ 即 $4|S|\log|S|$，为了防止数组越界，尽量开到不会`MLE`的最大节点数。

如果只需要判断一个区间是否有某个子串，把`cnt`改成`bool`然后修改`query`来减小常数。

```
int root[N];
struct MergeableSegmentTree {

    #define lch (tr[u].lc)
    #define rch (tr[u].rc)
    constexpr static int SZ=N*40;
    constexpr static int pos_l=0,pos_r=N-1;

    struct Node {
        int lc,rc;
        int cnt;
    } tr[SZ];
    int idx;

    int new_node() { return ++idx; }
```

```
    int merge(int x,int y) {
        if(!x||!y) return x|y;
        int u=new_node();
        lch=merge(tr[x].lc,tr[y].lc);
        rch=merge(tr[x].rc,tr[y].rc);
        tr[u].cnt=tr[lch].cnt+tr[rch].cnt;
        return u;
    }

    int __query(int u,int l,int r,int ql,int qr) {
        if(l>=ql&&r<=qr) return tr[u].cnt;
        int mid=l+r>>1;
        int res=0;
        if(lch&&mid>=ql) res+=__query(lch, l, mid, ql, qr);
        if(rch&&mid<qr) res+=__query(rch, mid+1, r, ql, qr);
        return res;
    }

    int query(int u,int ql,int qr) {
        if(ql>qr) return 0;
        return __query(u, pos_l, pos_r, ql, qr);
    }

    void __build(int &u,int l,int r,int p) {
        u=new_node();
        tr[u].cnt=1;
        if(l!=r) {
            int mid=l+r>>1;
            if(p<=mid) __build(lch,l,mid,p);
            else __build(rch,mid+1,r,p);
        }
    }

    void build(int &u,int p) { __build(u, pos_l, pos_r, p); }

    #undef lch
    #undef rch

} sgt;
```

按照拓扑序进行线段树合并。

```
void build_sgt(string &s) {
    for(int u=0,i=0;i<s.size();i++) {
        int c=s[i]-B;
        u=edp[u].ch[c];
        sgt.build(root[u], i);
    }
    for(int u:toporder) {
        int p=edp[u].link;
        if(p) root[p]=sgt.merge(root[p], root[u]);
```

```
    }
}
```

查询，为了排除半段字符串在外面的情况，`query`的时候的正确查询区间应为`[a+len-1,b]`。

```
int q;
cin>>q;
while(q--) {
    int l,r,a,b;
    cin>>l>>r>>a>>b;
    l--,r--,a--,b--;
    int len=r-l+1;
    int u=lca::find(l, r);
    cout<<sgt.query(root[u], a+len-1, b)<<endl;
}
```