ZZZZone的模板整理

ZZZZone的模板整理
黑科技
    快读
    long * long 取模
        1. 快速乘
        2. 黑科技
    unique实现 离散化
图论
    BFS求树的直径（未完成！！！！）
    二分图匹配（未完成！！！！）
    最小生成树
        Kruskal算法
数据结构
    归并排序
    并查集
    树状数组(Binary Indexed Tree, BIT)
    RMQ(Range Minimum/Maximum MaxQuery)
        ST表(Sparse Table)(静态)
        基于线段树(动态)
    线段树
    LCA
    划分树
    归并树
数学
    素数
        筛素数
    快速幂
动态规划
    LIS - 最长上升子序列O(NlogN)
    计数问题
字符串
    KMP
    AC自动机
几何
C++STL
    unique（去重）
    reverse
    vector
    next_permutation(下一个全排列)

lexicographical_compare（字典序比较）
merge （将两个已排序序列合并成新的排序序列）
inplace_merge （将两段排好序的序列归并排序）
builtin系列 （GCC）

黑科技

快读

```
namespace fastIO//输入外挂
{
    #define BUF_SIZE 100000
    //fread -> read
    bool IOerror = 0;
    inline char nc() {
        static char buf[BUF_SIZE], *p1 = buf + BUF_SIZE,
*pend = buf + BUF_SIZE;
        if(p1 == pend) {
            p1 = buf;
            pend = buf + fread(buf, 1, BUF_SIZE, stdin);
            if(pend == p1) {
                IOerror = 1;
                return -1;
            }
        }
        return *p1++;
    }
    inline bool blank(char ch) {
        return ch == ' ' || ch == '\n' || ch == '\r' || ch ==
'\t';
    }
    inline void read(int &x) {
        char ch;
        while(blank(ch = nc()));
        if(IOerror)
            return;
```

```
        for(x = ch - '0'; (ch = nc()) >= '0' && ch <= '9'; x = x * 10 + ch - '0');
    }
    #undef BUF_SIZE
};
using namespace fastIO;

void Out(int a)//输出外挂
{
    if(a>9)
        Out(a/10);
    putchar(a%10+'0');
}
```

*long * long 取模*

1. 快速乘

```
long long quickplus(long long  m,long long n,long long k)//返回m*n%k
{
    long long b = 0;
    while (n > 0)
    {
          if (n & 1)
             b = (b+m)%k;
          n = n >> 1 ;
          m = (m+m)%k;
    }
    return b;
}
```

2. 黑科技

> 大概意思就是强制转double然后搞一波再转回来。

```
LL mult( LL A, LL B, LL Mo )
{
    LL temp = ( ( LL ) ( ( db ) A*B/Mo+1e-6 ) * Mo );
    return A*B - temp;
}
```

unique实现 离散化

```
for(int i = 1; i <= n; i++) scanf("%d",&a[i]),b[i]=a[i];
sort(b+1,b+n+1);
int l=unique(b+1,b+n+1)-b-1;
fo(int i = 1; i <= n; i++) a[i]=lower_bound(b+1,b+l+1,a[i])-b;
```

图论

BFS求树的直径（未完成！！！！）

二分图匹配（未完成！！！！）

- 匈牙利算法-邻接矩阵

```
/*****************************************************************
//二分图匹配(匈牙利算法的DFS实现)(邻接矩阵形式)
//初始化:g[][]两边顶点的划分情况
//建立g[i][j]表示i->j的有向边就可以了，是左边向右边的匹配
//g没有边相连则初始化为0
//uN是匹配左边的顶点数，vN是匹配右边的顶点数
//调用:res=hungary();输出最大匹配数
//优点:适用于稠密图，DFS找增广路，实现简洁易于理解
//时间复杂度:O(VE)
```

```
10 ************************************************************
   /
11
12
13 const int MAXN = 510;
14 int edge[MAXN][MAXN];//邻接矩阵
15 int linker[MAXN];
16 bool used[MAXN];
17
18 bool dfs(int u){
19     for(int v = 1; v <= n; v++){
20         if(edge[u][v] && !used[v]){
21             used[v] = true;
22             if(linker[v] == -1 || dfs(linker[v])){
23                 linker[v] = u;
24                 return true;
25             }
26         }
27     }
28     return false;
29 }
30
31 int lungary(){
32     int res = 0;
33     memset(linker, -1, sizeof(linker));
34     for(int i = 1; i <= n; i++){
35         memset(used, false, sizeof(used));
36         if(dfs(i)) res++;
37     }
38     return res;
39 }
```

最小生成树

Kruskal算法

复杂度$O(M * log(M))$

```
1 •const int MaxN = 20000;
```

```
2
3  struct EDGE {
4      int u, v, w;
5      void Make(int x, int y, int t) {
6          u = x; v = y; w = t;
7      }
8  }edge[MaxN + 5];
9  int n, tot, father[MaxN + 5];
10
11 int Find(int x)
12 {
13     if (father[x] == x) return x;
14     return father[x] = Find(father[x]);
15 }
16
17 void Init()
18 {
19     int a, b, w; tot = 0;
20     for (int i = 1; i <= n; i++) father[i] = i;
21     for (int i = 1; i <= n * (n - 1) / 2; i++) {
22         scanf("%d%d%d", &a, &b, &w);
23         edge[++tot].Make(a, b, w);
24     }
25 }
26
27 bool cmp(EDGE x, EDGE y) {return x.w < y.w;}
28
29 void Solve()
30 {
31     int Mst_ans = 0;
32     sort(edge + 1, edge + tot + 1, cmp);
33     for (int i = 1; i <= tot; i++) {
34         int u = edge[i].u, v = edge[i].v;
35         if (Find(u) != Find(v)) {
36             father[Find(u)] = v;
37             Mst_ans += edge[i].w;
38         }
39     }
40     printf("%d\n", Mst_ans);
41 }
```

数据结构

归并排序

```
//参考紫书算法竞赛入门经典
void merge_sort(int *A, int l, int r, int *T){ //[l, r) 排序.外部调用区间为[0, n)
    if(r - l > 1){
        int mid = l+(r-l)/2;
        int p = l, q = mid, now = l;
        // 对左右两部分区间分别归并排序
        merge_sort(A, l, mid, T);
        merge_sort(A, mid, r, T);
        // 合并左右两部分
        while(p < mid || q < r){
            if(q >= r ||  (p < mid && A[p] <= A[q])){
                T[now++] = A[p++];
            }
            else{
                T[now++] = A[q++];
                cnt += mid - p; // cnt记录的是逆序对个数
            }
        }
        for(int i = l; i < r; i++) A[i] = T[i];
    }
}
```

并查集

```cpp
int Find(int id)
{
    if(id == fa[id]) return id;
    else return fa[id] = Find(fa[id]);
}
void addu(int u, int v)
{
    int x = Find(u);
    int y = Find(v);
    if(x != y)    fa[x] = y;
}
```

树状数组(Binary Indexed Tree, BIT)

树状数组可以进行的操作:

1. 单点修改， 区间查询
2. 区间修改， 单点查询

一维:

```cpp
const int MAXN = 1e5;

inline int lowbit(int x) {return (x & -x); }

void Add(int p, int value){
    for(int i = p; i <= MAXN; i += lowbit(i))
        presum[i] += value;
}

int getsum(int p){
    int sum = 0;
    for(int i = p; i > 0; i -= lowbit(i))
        sum += presum[i];
    return sum;
}

```

二维:

```
inline int lowbit(int x) {return (x & -x); }

void add(int x, int y, int num, int p){
    for(int i = x; i <= MAXN; i += lowbit(i))
        for(int j = y; j <= MAXN; j+= lowbit(j))
            a[i][j][p] += num;
}

int getsum(int x, int y, int p){
    int ans = 0;
    for(int i = x; i > 0; i -= lowbit(i))
        for(int j = y; j > 0; j -= lowbit(j))
            ans += a[i][j][p];
    return ans;
}

```

RMQ(Range Minimum/Maximum MaxQuery)

ST表(Sparse Table)(静态)

一维:

```
int mx[MAXN+5][25];
int mn[MAXN+5][25];
int Log2[MAXN+5];
//预处理O(nlog(n))  查询O(1)
struct RMQ{
    void init(){
        Log2[0] = -1;
        for(int i = 1; i <= n; i++) Log2[i] = Log2[i >> 1] +
1;
        for(int i = 1; i <= n; i++) mx[i][0] = mn[i][0] =
num[i];
        for(int j = 1; (1 << j) <= n; j++){
            for(int i = 1; i + (1 << j) - 1 <= n; i++){
                mx[i][j] = max(mx[i][j-1], mx[i+(1<<(j-1))]
[j-1]);
```

```
                mn[i][j] = min(mn[i][j-1], mn[i+(1<<(j-1))][j-1]);
            }
        }
    }
    int query_max(int ql, int qr){
        int k = Log2[qr-ql+1];
        return max(mx[ql][k], mx[qr-(1<<k)+1][k]);
    }// 查询区间[ql, ql+2^k-1] 和 [qr-2^k+1, qr];
    int query_min(int ql, int qr){
        int k = Log2[qr-ql+1];
        return min(mn[ql][k], mn[qr-(1<<k)+1][k]);
    }
}rmq;
```

二维:

```
struct RMQ_2D{
    void init(){
        for(int i = 1; i <= n; i++)
            for(int j = 1; j <= m; j++)
                dpmax[i][j][0][0] = a[i][j];
        Log2[0] = -1;
        for(int i = 1; i <= MAXN; i++) Log2[i] = Log2[i >> 1] + 1;
        for(int i = 0; (1 << i) <= n; i++){
            for(int j = 0; (1 << j) <= m; j++){
                if(i == 0 && j == 0) continue;
                for(int r=1;r+(1<<i)-1<=n;r++){
                    for(int c=1;c+(1<<j)-1<=m;c++){
                        if(i==0)
                            dpmax[r][c][i][j]=max(dpmax[r][c][i][j-1],dpmax[r][c+(1<<(j-1))][i][j-1]);
                        else
                            dpmax[r][c][i][j]=max(dpmax[r][c][i-1][j],dpmax[r+(1<<(i-1))][c][i-1][j]);
                    }
                }
            }
        }
    }

    int query_max(int r1, int c1, int r2, int c2){
        int kr = Log2[r2-r1+1];
```

```
        int kc = Log2[c2-c1+1];
        int t1=dpmax[r1][c1][kr][kc];
        int t2=dpmax[r2-(1<<kr)+1][c1][kr][kc];
        int t3=dpmax[r1][c2-(1<<kc)+1][kr][kc];
        int t4=dpmax[r2-(1<<kr)+1][c2-(1<<kc)+1][kr][kc];
        return max(max(t1,t2),max(t3,t4));
    }

}rmq;
```

基于线段树(动态)

一维:

```
// 此模板为查询最小值。 最大值同理修改即可。
// 挑战程序设计竞赛模板   //下标从0开始。
// 预处理O(n)  查询O(log(n))
const int MAXN = 1 << 17;
const int INF_MAX = 1 << 30;

int n, dat[2 * MAXN -1];

void init(int n_){ // 初始化n扩大到2的幂次。
    n = 1;
    while(n < n_) n *= 2;
    for(int i = 0; i < 2 * n - 1; i++) dat[i] = INF_MAX;
}

void update(int k, int num){// 向上
    k += n - 1;
    dat[k] = num;
    while(k > 0){
        k = (k-1) / 2;
        dat[k] = min(dat[k*2+1], dat[k*2+2]);
    }
}
// 求[a, b)最值。 k是节点编号。  左闭右开
// 外部调用query(a, b, 0, 0, n);
int query(int a, int b, int k, int l, int r){
    if(r <= a || b <= l) return INF_MAX;
    if(a <= l &&r <= b) return dat[k];
    else{

```

```
        int vl = query(a, b, k*2+1, l, (l+r)>>1);
        int vr = query(a, b, k*2+2, (l+r)>>1, r);
        return min(vl, vr);
    }
}

```

线段树

```
typedef long long LL;
const int MaxN = 1e5 + 5;
LL sum[4 * MaxN];
LL add[4 * MaxN];
int n, m;

void pushup(int rt){    //向上更新
    sum[rt] = sum[rt << 1] + sum[rt << 1 | 1];
}

// 初始化      build(1, n, 1);
void build(int l, int r, int rt){
    if(l == r){
        scanf("%lld", &sum[rt]);
        return;
    }
    int mid = (l + r) >> 1;
    build(l, mid, rt << 1);
    build(mid + 1, r, rt << 1 | 1);
    pushup(rt);
}

void pushdown(int rt, int len){  // 核心 不得不往下的时候再下放
    if(add[rt]){
        add[rt << 1] += add[rt];
        add[rt << 1 | 1] += add[rt];
        sum[rt << 1] += (len - (len >> 1)) * add[rt];
        sum[rt << 1 | 1] += (len >> 1) * add[rt];
        add[rt] = 0;
    }
}
```

```cpp
//外部调用update(l, r, x, 1, n, 1);
void update(int L, int R, int num, int l, int r, int rt){  //更新区间
    if(L <= l && R >= r){
        add[rt] += num;
        sum[rt] += (LL)num * (r - l + 1);
        return;
    }
    pushdown(rt, r - l + 1);
    int mid = (l + r) >> 1;
    if(L <= mid) update(L, R, num, l, mid, rt << 1);
    if(R > mid) update(L, R, num, mid + 1, r, rt << 1 | 1);
    pushup(rt);
}
// 外部调用 query(l, r, 1, n, 1));
LL query(int L, int R, int l, int r, int rt){
    if(L <= l && R >= r) return sum[rt];
    pushdown(rt, r - l + 1);
    int mid = (r + l) >> 1;
    LL ret = 0;
    if(L <= mid) ret += query(L, R, l, mid, rt << 1);
    if(R > mid) ret += query(L, R, mid + 1, r, rt << 1 | 1);
    return ret;
}

```

LCA

```cpp
int fa[MaxN], dis[MaxN];
int up[MaxN][20];
vector<int> edge[MaxN];
int n, q;
/* 先调用dfs(root, 1), 在调用lca_init(), 最后lca(u, v) 查询 */
// up[i][j] 表示i点往上2^j的点是谁。
void dfs(int x, int d){
    dis[x] = d;
    for(int i = 0; i < edge[x].size(); i++){
        if(!dis[edge[x][i]]){
            up[edge[x][i]][0] = x;
            dfs(edge[x][i], d + 1);
```

```
        }
    }
}
void lca_init(){
    for(int j = 1; (1 << j) <= n; j++)
        for(int i = 1; i <= n; i++)
            up[i][j] = up[up[i][j - 1]][j - 1];
}
int lca(int a, int b){
    if(dis[a] < dis[b]) swap(a, b); // a在下面
    int c = dis[a] - dis[b];
    for(int i = 0; (1 << i) <= n; i++)  // 比如c 是5 (101)  那么先走2^0, 再走2^2
        if(c & (1 << i)) a = up[a][i];
    if(a == b) return a;
    for(int i = 14; i >= 0; i--){
        if(up[a][i] != up[b][i]){
            a = up[a][i];
            b = up[b][i];
        }
    }
    return up[a][0];
}
```

划分树

在$O(log(n))$复杂度下查询区间第k大。

```
const int MaxN = 1e5, Pow = 20;

int n, m, k;
int a[MaxN + 5];
int tree[Pow + 5][MaxN + 5];
int toleft[Pow + 5][MaxN + 5];
int sorted[MaxN + 5];

bool cmp(int x, int y) {return x < y;}


// 初始化区间[l,r]
```

```cpp
void Build(int dep, int l, int r) {
    if(l == r) return;
    int mid = (l + r) >> 1;
    int same = mid - l + 1;
    for(int i = l; i <= r; i++)
        if(tree[dep][i] < sorted[mid]) same--;
    int lpos = l;
    int rpos = mid + 1;
    for(int i = l; i <= r; i++) {
        if(tree[dep][i] < sorted[mid]) {// 小的值直接放到左儿子
            tree[dep + 1][lpos++] = tree[dep][i];
        }
        else if(tree[dep][i] == sorted[mid] && same > 0) { // 左儿子还有空能放相同的数
            tree[dep + 1][lpos++] = tree[dep][i]; same--;
        }
        else tree[dep + 1][rpos++] = tree[dep][i]; // 剩下的大于等于的放到右儿子
        toleft[dep][i] = toleft[dep][l - 1] + lpos - l; // 更新有多少个数放到了左边
    }
    Build(dep + 1, l, mid);
    Build(dep + 1, mid + 1, r);
}

// 询问[ql, qr] 中第k大的数
// 外部调用query(1, 1, n, l, r, k);
int query(int dep, int L, int R, int ql, int qr, int k) {
    if(ql == qr) return tree[dep][ql];
    int mid = (L + R) >> 1;
    int cnt = toleft[dep][qr] - toleft[dep][ql - 1];
    if(cnt >= k) {
        int newl = L + toleft[dep][ql - 1] - toleft[dep][L - 1];
        int newr = newl + cnt - 1;
        return query(dep + 1, L, mid, newl, newr, k);
    }
    else {
        int newr = qr + toleft[dep][R] - toleft[dep][qr];
        int newl = newr - (qr - ql - cnt);
        return query(dep + 1, mid + 1, R, newl, newr, k - cnt);
    }
}

```

```
void Init() {
    for(int i = 1; i <= n; i++) {
        scanf("%d", &a[i]);
        sorted[i] = tree[1][i] = a[i];
    }
    sort(sorted + 1, sorted + n + 1, cmp);
    Build(1, 1, n);
}

```

归并树

n个数， m次询问. 查询区间第k大复杂度为$O(nlog(n)+m*log^3(n))$.

```
int n, m;
int a[MAXN+5], num[MAXN+5];
vector<int> dat[ST_SIZE];

void init(int k, int l, int r){
    if(r - l == 1){
        dat[k].push_back(a[l]);
    }
    else{
        int lch = k * 2 + 1, rch = k * 2 + 2;
        init(lch, l, (l+r)/2);
        init(rch, (l+r)/2, r);
        dat[k].resize(r-l);
        merge(dat[lch].begin(), dat[lch].end(),
dat[rch].begin(), dat[rch].end(), dat[k].begin());
        // 利用STL自带的merge函数把两个儿子的数列合并
    }
}

// 计算[ql, qr) 中不超过x的个数
// k是节点的编号, 对应区间[l, r)
int query(int ql, int qr, int x, int k, int l, int r){
    if(qr <= l || r <= ql){//完全不相交
        return 0;
    }
    else if(ql <= l && r <= qr){ // 询问完全包含当前区间
```

```
26          return upper_bound(dat[k].begin(), dat[k].end(), x) -
    dat[k].begin();
27      }
28      else{
29          int lcnt = query(ql, qr, x, k*2+1, l, (l+r)/2);
30          int rcnt = query(ql, qr, x, k*2+2, (l+r)/2, r);
31          return lcnt + rcnt;
32      }
33  }
34
35  int main()
36  {
37      scanf("%d %d", &n, &m);
38      for(int i = 0; i < n; i++){
39          scanf("%d", &a[i]);
40          num[i] = a[i];
41      }
42      init(0, 0, n);
43      sort(num, num + n); // 排好序的数组，  方便二分
44      while(m--){
45          int l, r, k;
46          scanf("%d %d %d", &l, &r, &k);
47          l--; // 查询区间[l, r]第k大，  转化为下标为零开始的区间[l,
    r)
48          int lb = -1, ub = n-1;
49          while(ub - lb > 1){ // 二分k大的数进行check
50              int mid = (ub + lb) / 2;
51              int c = query(l, r, num[mid], 0, 0, n);
52              if(c >= k) ub = mid;
53              else lb = mid;
54          }
55          printf("%d\n", num[ub]);
56      }
57      return 0;
58  }
```

数学

素数

筛素数

```
//valid[i]表示i是否为素数。
//ans为素数表。
//tot是素数总数。

/* O(NlogN) */
void getPrime(int n, int &tot, int ans[]){
    tot = 0;
    memset(valid, true, sizeof(valid));
    for(int i = 2; i <= n; i++) if(valid[i]){
        if(n/i<i) break; // 防止i*i爆int
        for(int j = i*i; j <= n; j += i) valid[j] = false;
    }
    for(int i = 2; i <= n; i++) if(valid[i]) ans[++tot] = i;
}

/* O(N) */
void getPrime(int n, int &tot, int ans[]){
    memset(valid, true, sizeof(valid));
    for(int i = 2; i <= n; i++){
        if(valid[i]) ans[++tot] = i;
        for(int j = 1; j <= tot && i * ans[j] <= n; j++){
            valid[i*ans[j]] = false;
            if(i%ans[j] == 0) break;
        }
    }
}

```

快速幂

```cpp
// log(y)复杂度求x^y%mod;
LL fast_pow(LL x, int y){
    LL ans = 1LL;
    while(y != 0){
        if(y & 1) ans = (ans*x) % mod;
        y >>= 1;
        x = (x * x)%mod;
    }
    return ans % mod;
}
```

动态规划

LIS - 最长上升子序列O(NlogN)

```cpp
int dp[MAXN+5];

void solve(){
    fill(dp, dp + n, INF);
    for(int i = 0; i < n; i++) {
        *lower_bound(dp, dp + n, a[i]) = a[i];
    }
    printf("%d\n", lower_bound(dp, dp + n, INF) - dp);
}
```

计数问题

将n划分成不超过m组

> 白书模板

```cpp
int dp[MAXN+5][MAXN+5];
int n, m, Mod;
//dp[i][j]表示j的i划分的总数
```

```
void solve(){
    dp[0][0] = 1;
    for(int i = 1; i <= m; i++){
        for(int j = 0; j <= n; j++){
            if(j - i >= 0)
                dp[i][j] = (dp[i-1][j] + dp[i][j-i]) % Mod;
            else
                dp[i][j] = dp[i-1][j];
        }
    }
    printf("%d\n", dp[m][n]);
}
```

字符串

KMP

```
void Kmp(char s[], char t[], int &cnt)
{
    int p = -1, len1 = strlen(s), len2 = strlen(t);
    nxt[0] = -1;
    for (int i = 1; i < len2; i++) {
        while (p > -1 && t[p + 1] != t[i]) p = nxt[p];
        if (t[p + 1] == t[i]) p++;
        nxt[i] = p;
    }
    p = -1;
    for (int i = 0; i < len1; i++) {
        while (p > -1 && s[i] != t[p + 1]) p = nxt[p];
        if (s[i] == t[p + 1]) p++;
        if (p == len2 - 1) cnt++, p = nxt[p];
    }
}
```

AC自动机

```cpp
#include<cstdio>
#include<cstring>
#include<cmath>
#include<cstdlib>
#include<algorithm>
#include<queue>
using namespace std;
const int MaxN = 1000005;
const int MaxV = 1000005;

int n;
int nxt[MaxN][30], fail[MaxN], edd[MaxN], root, L;
//nxt记录节点，edd当前字符串结尾个数，fail失配指针。
int mark[MaxN];
char buf[MaxN], s[MaxV];
//裸模板跑873ms    优化的跑了280ms 交的g++   c++TLE

/*
构造失败指针的过程概括起来就一句话：设这个节点上的字母为C，沿着他父亲的失败指针走，直到走到一个节点，他的儿子中也有字母为C的节点。然后把当前节点的失败指针指向那个字母也为C的儿子。如果一直走到了root都没找到，那就把失败指针指向root。
*/

int newnode(){
    for(int i = 0; i < 26; i++) nxt[L][i] = -1; // 节点链接的初始化为-1
    edd[L] = 0;
    mark[L] = 0;
    return L++;
}

void init(){
    L = 0;
    root = newnode();
}

void insert(char buf[], int len){
    int now = root;
    for(int i = 0; i < len; i++){
        if(nxt[now][buf[i] - 'a'] == -1) nxt[now][buf[i] - 'a'] = newnode();
        now = nxt[now][buf[i] - 'a'];
    }
```

```
    edd[now]++;
}

void build(){
    queue<int> que;
    for(int i = 0; i < 26; i++){
        if(nxt[root][i] == -1) nxt[root][i] = root;
        else{
            fail[nxt[root][i]] = root;
            que.push(nxt[root][i]);
            //如果有连边，则将节点插入队列，并将fail指针指向root
        }
    }
    while(!que.empty()){
        int now = que.front();
        que.pop();
        for(int i = 0; i < 26; i++){
            //没有连边，将该边指向当前节点fail指针指向的对应的节点。
            if(nxt[now][i] == -1) nxt[now][i] =
nxt[fail[now]][i];
            else{
                //有连边，将儿子节点的fail指针指向当前节点fail指针
对应的节点。
                fail[nxt[now][i]] = nxt[fail[now]][i];
                que.push(nxt[now][i]);
                //加入队列继续遍历//
            }
        }
    }
}

int query(char buf[], int len){
    int now = root;
    int res = 0;
    for(int i = 0; i < len; i++){
        now = nxt[now][buf[i] - 'a'];
        int temp = now;
        //这里优化访问过的就不再访问
        while(temp != root && mark[temp] == 0){
            res += edd[temp];
            edd[temp] = 0;
            mark[temp] = 1;
            temp = fail[temp];
        }
        /*
```

```
        //裸模板:
          while(temp != root){
            res += edd[temp];
            edd[temp] = 0;//模式串在主串中匹配一次就可以了
            temp = fail[temp];
          }
        */
    }
    return res;
}

int main(){
    int T;
    scanf("%d", &T);
    while(T--){
        scanf("%d", &n);
        init();
        int Maxlen = 0;
        for(int i = 1; i <= n; i++){
            scanf("%s", buf);
            int len = strlen(buf);
            insert(buf, len);
        }
        scanf("%s", s);
        Maxlen = strlen(s);
        build();
        int tot = query(s, Maxlen);
        printf("%d\n", tot);
    }
    return 0;
}
```

几何

C++STL

unique（去重）

需要头文件： #include<algorhtm> 作用：“去掉”容器中 相邻元素的重复元素（不一定要求数组有序），它会把重复的元素添加到容器末尾（所以数组大小并没有改变），而返回值是去重之后的尾地址。

```
//用来去掉相邻元素个数后剩余的个数。
// 一般来说先sort排序，  再进行去重。
sz = unique(b + 1,b + n + 1)-(b + 1); // 下标从1开始
sz = unique(a,a + n) - a;  //下标从0开始
```

如果要删去重复元素， 可以erase尾部元素， 或者直接改变大小。

```
for (int i = 0;i < n;++i) scanf("%d",&a[i]);
sort (a, a + n);
vector<int>v (a, a + n);
vector<int>::iterator it = unique (v.begin(), v.end() );   //迭代器
v.erase (it, v.end() );//这里就是把后面藏起来的重复元素删除了
```

结构体**struct**使用**unique**： 方法和sort排序的cmp一样， 也可是通过结构体的重载==号。

```
struct Point{
    int x, y;
    bool friend operator == (Point a, Point b){
        if(a.x != b.x) return a.y == b.y;
        return a.x == b.x;
    }
}b[10];
...

int m = unique(b, b + n) - b; // 重载==


//********************
struct Point{
    int x, y;
}b[10];

bool uni(Point a, Point b){
```

```
    if(a.x != b.x) return a.y == b.y;
    return a.x == b.x;
}
...

int m = unique(b, b + n, uni) - b;//类似于sort的cmp函数。

```

reverse

reverse(beg,end) 将区间[beg,end)内的元素全部逆序； 所需头文件 #include<algorithm> 复杂度 线性

只能传入迭代器。

```
普通数组:
scanf("%d", &n);
for(int i = 0; i < n; i++) scanf("%d", &a[i]);
sort(a, a + n);
reverse(begin(a), begin(a)+n);
// 这里最好不要使用reverse(begin(a), end(a)); 因为end(a)是数组尾部， 平时acm中我们经常会+5多开一些空间，就无法正确反转了。
for(int i = 0 ; i < n; i++) printf("%d ", a[i]);
printf("\n");
//**********  vector
for(int i = 0; i < n; i++) b.push_back(a[i]);
reverse(b.begin(), b.end());
for(int i = 0 ; i < n; i++) printf("%d ", b[i]);
printf("\n");

```

vector

二维vector 声明： vector<vector<int> > ans; 添加元素:

```
for(int i = 1; i <= n; i++){
    ans.push_back(vector<int>(0, 0));
    // 先push_back一个空的一维vector
    // vector<int>(x, y) 表示创建一个有x个y的一维数组。
    for(int j = 1; j <= m; j++){
        ans[i].push_back(i*j);
    }
}
```

next_permutation(下一个全排列)

需要头文件：#include<algorithm> 用法：next_permutation(arrays,arrays+n); 用途: 长度为n的数组arrays生成下一个全排列。 同理， 相应的有**prev_permutation**（上一个全排列）。

lexicographical_compare（字典序比较）

所需头文件: #include<algorithm> 用法: lexicographical_compare(s1, s2) 用途: 比较s1和s2的字典序， s1比s2小返回true， 反之返回false。

merge （将两个已排序序列合并成新的排序序列）

函数接口：

```
std::merge (first1,last1,first2,last2, first3);
```

将有序的序列[first1, last1), [first2, last2)合并后放到first3开始的位置. 复杂度：O(len1 + len2);

使用样例：

```cpp
// merge algorithm example
#include <iostream>     // std::cout
#include <algorithm>    // std::merge, std::sort
#include <vector>       // std::vector

int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  std::vector<int> v(10);

  std::sort (first,first+5);
  std::sort (second,second+5);
  std::merge (first,first+5,second,second+5,v.begin());

  std::cout << "The resulting vector contains:";
  for (std::vector<int>::iterator it=v.begin(); it!=v.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

源码:

```cpp
// 可以cmp ,  也可以重载 <
template <class InputIterator1, class InputIterator2, class OutputIterator>
  OutputIterator merge (InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2,
                        OutputIterator result)
{
  while (true) {
    if (first1==last1) return std::copy(first2,last2,result);
    if (first2==last2) return std::copy(first1,last1,result);
    *result++ = (*first2<*first1)? *first2++ : *first1++;
  }
}
```

inplace_merge （将两段排好序的序列归并排序）

函数接口：

```
void inplace_merge (first, middle, last);
```

将两个已排序的序列[first,middle)和[middle,last)合并成单一有序序列.

使用样例：

```
#include <iostream>     // std::cout
#include <algorithm>    // std::inplace_merge, std::sort, std::copy
#include <vector>       // std::vector

int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  std::vector<int> v(10);
  std::vector<int>::iterator it;

  std::sort (first,first+5);
  std::sort (second,second+5);

  it=std::copy (first, first+5, v.begin());
     std::copy (second,second+5,it);

  std::inplace_merge (v.begin(),v.begin()+5,v.end());

  std::cout << "The resulting vector contains:";
  for (it=v.begin(); it!=v.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

builtin系列 （GCC）

__builtin_ffs(x)：x中最后一个为1的位是从后向前的第几位。**__builtin_popcount(x)**：x中1的个数。**__builtin_parity(x)**：x中1的奇偶性。