基础算法合集

2025-7-3

目录

一、经典数据结构

1. 并查集

并查集是一种用于管理元素所属集合的数据结构，实现为一个森林，其中每棵树表示一个集合，树中的节点表示对应集合中的元素。

优化方法有：1.按秩合并，秩是当前树的最大高度，按秩合并即将高度小的树合并到高度大的树上，以尽可能压低最终树高。2.路径压缩：每次查询时，将查询路径上的点的父亲，设置为该树的根节点，以压低树高。综合使用两种策略，能将 find 和 union 的均摊复杂度压到 O（1）。

```
inline void init_set(int n) //初始化并查集
{
    for (int i = 1; i <= n; i++)
        s[i] = i;
}
int find_set(int x) //查找
{
    if (x != s[x])
        s[x] = find_set(s[x]);
    return s[x];
}
void union_set(int x, int y) //原始合并
{
    x = find_set(x);
    y = find_set(y);
    if (x != y)
        s[x] = s[y],
            flag[xx]=flag[yy]=(flag[xx]||flag[yy]); //染色并查集，用于隔离
}
```

1-1. 种类并查集（实现同上）

种类并查集是一种特殊的并查集，对于每个属性点，都维护其“反点”。种类并查集可用于判断一系列等于和不等于的关系中，是否存在矛盾。

```
for(int i=1;i<=m;i++){
    if(q[i]==0){//种类相同
        if(find(mm[l[i]])==find(mm[r[i]])+cnt){
            cout<<i-1;
            return 0;
        }
        join(mm[l[i]],mm[r[i]]);
        join(mm[l[i]]+cnt,mm[r[i]]+cnt);
    }
    else{//种类不同
        if(find(mm[l[i]])==find(mm[r[i]])){
            cout<<i-1;
            return 0;
        }
        join(mm[l[i]],mm[r[i]]+cnt);
        join(mm[l[i]]+cnt,mm[r[i]]);
    }

}
```

2. 线段树

线段树可以在 O(log N) 的时间复杂度内实现单点修改、区间修改、区间查询（区间求和，求区间最大值，求区间最小值）等操作。但线段树的码量略大，可使用 ST 表、树状数组作为其简化版本。**建树的复杂度为 O（n）**，小于 ST 表。

线段树将每个长度不为 1 的区间划分成左右两个区间递归求解（**可差分**），若达到修改或查询的目的区间时,则停止递归并修改或查询该节点,并向上合并修改或答案(**可合并**)。

线段树使用一个 lazy_tag，来维护上述停止递归时的修改情况。以在下次更深层次的修改或查询时，完成对该节点的子节点的修改。

复杂度保证：考虑到一个完整的区间，被递归为左右两个区间的次数不超过 logn。

```
long long sum[400005];
long long tag[400005];
void push_up(int now){
    sum[now]=sum[2*now]+sum[2*now+1];  //求区间和
    sum[now]=max(sum[2*now],sum[2*now+1]); //求最大值
```

```
}
void build(int now,int l,int r){
    if(l==r){
        scanf("%lld",&sum[now]);
        return;
    }
    int mid=(l+r)/2;
    build(now*2,l,mid);
    build(now*2+1,mid+1,r);
    push_up(now);
}
void spread(int now,int l,int r){
    if(tag[now]){
        tag[now*2]+=tag[now];
        tag[now*2+1]+=tag[now];
        int mid=(l+r)/2;
        sum[now*2]+=tag[now]*(mid-l+1);
        sum[now*2+1]+=tag[now]*(r-mid);
        tag[now]=0;
    }
}
void change(int now,int x,int y,int l,int r,int t){
    if(x<=l&&r<=y){
        sum[now]+=t*(r-l+1); //求和
        tag[now]+=t; //求和
        return;
    }
    spread(now,l,r);
    int mid=(l+r)/2;
    if(x<=mid) change(2*now,x,y,l,mid,t);
    if(y>mid) change(2*now+1,x,y,mid+1,r,t);
    push_up(now);
}
long long ask(int now,int x,int y,int l,int r){
    if(x<=l&&r<=y){
        return sum[now];
    }
    spread(now,l,r);
    int mid=(l+r)/2;
    long long ans=0;
    if(x<=mid) ans+=ask(now*2,x,y,l,mid);  //求和
    if(y>mid) ans+=ask(now*2+1,x,y,mid+1,r); //求和
if(x<=mid) ans=max(ans,ask(now*2,x,y,l,mid));  //求最大值
if(y>mid) ans=max(ans,ask(now*2+1,x,y,mid+1,r)); //求最大值
```

```
    return ans;
}
```

2-1 乘法、加法线段树

```
void spread(int now,int l,int r){ //乘法优先级比加法高，故先处理乘法 tag
    if(mul[now]!=1){
        mul[now*2]*=mul[now];
        mul[now*2]%=mm;
        mul[now*2+1]*=mul[now];
        mul[now*2+1]%=mm;
        tag[now*2]*=mul[now];
        tag[now*2+1]*=mul[now];
        tag[now*2]%=mm;
        tag[now*2+1]%=mm;
        sum[now*2]= (sum[now*2]*mul[now])%mm;
        sum[now*2+1]=(sum[now*2+1]*mul[now])%mm;
        mul[now]=1;
    }
    if(tag[now]!=0){
        tag[now*2]+=tag[now];
        tag[now*2+1]+=tag[now];
        tag[now*2]%=mm;
        tag[now*2+1]%=mm;
        int mid=(r+l)/2;
        sum[now*2]+=tag[now]*(mid-l+1);
        sum[now*2]%=mm;
        sum[now*2+1]+=tag[now]*(r-mid);
        sum[now*2+1]%=mm;
        tag[now]=0;
    }
}
void add(int now,int x,int y,int l,int r,int t){
    if(x<=l&&r<=y){
        sum[now]+=t*(r-l+1);
        sum[now]%=mm;
        tag[now]+=t;
        tag[now]%=mm;
        return;
    }
    spread(now,l,r);
```

```
    int mid=(l+r)/2;
    if(x<=mid) add (2*now,x,y,l,mid,t);
    if(y>mid) add (2*now+1,x,y,mid+1,r,t);
    push_up(now);
}

void mul_(int now,int x,int y,int l,int r,int t){
    if(x<=l&&r<=y){
        sum[now]*=t;
        sum[now]%=mm;
        mul[now]*=t;
        mul[now]%=mm;
        tag[now]*=t;
        tag[now]%=mm;
        return;
    }
    spread(now,l,r);
    int mid=(l+r)/2;
    if(x<=mid) mul_(2*now,x,y,l,mid,t);
    if(y>mid)  mul_(2*now+1,x,y,mid+1,r,t);
    push_up(now);
}
```

2-2.典例：小白逛公园

小白只可以选择第 a 个和第 b 个公园之间（包括 a,b 两个公园）选择**连续**的一些公园玩。且这些公园的分数之和尽可能高。

```
struct node{
    long long sum;
    long long mx;
    long long L;
    long long R;
} tree[2000005];

void push_up(int now){
    tree[now].mx=tree[now*2].R+tree[now*2+1].L;
    tree[now].mx=max(tree[now].mx,tree[now*2].R);
    tree[now].mx=max(tree[now].mx,tree[now*2+1].L);
    tree[now].mx=max(tree[now].mx,tree[now*2].mx);
    tree[now].mx=max(tree[now].mx,tree[now*2+1].mx);
    tree[now].R=max(tree[now*2+1].R,tree[now*2+1].sum+tree[now*2].R);
    tree[now].L=max(tree[now*2].L,tree[now*2].sum+tree[now*2+1].L);
```

```
    tree[now].sum=tree[now*2].sum+tree[now*2+1].sum;
}
void build(int now,int l,int r){
    if(l==r){
        scanf("%lld",&tree[now].mx);
        tree[now].sum=tree[now].R=tree[now].L=tree[now].mx;
        return;
    }
    int mid=(l+r)/2;
    build(now*2,l,mid);
    build(now*2+1,mid+1,r);
    push_up(now);
}
void change(int now,int x,int y,int l,int r,int t){
    if(l==r){//单点修改
        tree[now].sum=tree[now].R=tree[now].L=tree[now].mx=t;
        return;
    }

    int mid=(l+r)/2;
    if(x<=mid) change(2*now,x,y,l,mid,t);
    if(y>mid) change(2*now+1,x,y,mid+1,r,t);
    push_up(now);
}
node ask(int now,int x,int y,int l,int r){
    if(x<=l&&r<=y){
        return tree[now];
    }
    int mid=(l+r)/2;

//注意分类:
    if(y<=mid) return ask(now*2,x,y,l,mid); //完整段
    else if(x>mid) return ask(now*2+1,x,y,mid+1,r); //完整段
    else{
        node t;
        node ll=ask(now*2,x,y,l,mid);
        node rr=ask(now*2+1,x,y,mid+1,r);
        t.mx=ll.R+rr.L;
        t.mx=max(t.mx,ll.R);
        t.mx=max(t.mx,rr.L);
        t.mx=max(t.mx,ll.mx);
        t.mx=max(t.mx,rr.mx);
        t.R=max(rr.R,rr.sum+ll.R);
        t.L=max(ll.L,ll.sum+rr.L);
```

```
        t.sum=ll.sum+rr.sum;
        return t;
    }
}
```

2-3.典例：求区间内 X²的和

```
double sum[400005];

 double  sum2[400005];

 double tag[400005];
void push_up(int now){
    sum[now]=sum[2*now]+sum[2*now+1];
    sum2[now]=sum2[2*now]+sum2[2*now+1];
}
void build(int now,int l,int r){

    if(l==r){
        scanf("%lf",&sum[now]);
        sum2[now]=sum[now]*sum[now];
        return;
    }
    int mid=(l+r)/2;
    build(now*2,l,mid);
    build(now*2+1,mid+1,r);
    push_up(now);
}
void spread(int now,int l,int r){
    if(fabs(tag[now]-0)>0.00000000001){
        tag[now*2]+=tag[now];
        tag[now*2+1]+=tag[now];
        int mid=(l+r)/2;
        sum2[now*2]+=2*tag[now]*sum[now*2]+(mid-l+1)*tag[now]*tag[now];
        sum2[now*2+1]+=2*tag[now]*sum[now*2+1]+(r-mid)*tag[now]*tag[now];
        sum[now*2]+=tag[now]*(mid-l+1);
        sum[now*2+1]+=tag[now]*(r-mid);

        tag[now]=0;
    }
}
void change(int now,int x,int y,int l,int r,double t){
```

```
    if(x<=l&&r<=y){
        sum2[now]+=2*t*sum[now]+(r-l+1)*t*t;
        sum[now]+=t*(r-l+1);
        tag[now]+=t;
        return;
    }
    spread(now,l,r);
    int mid=(l+r)/2;
    if(x<=mid) change(2*now,x,y,l,mid,t);
    if(y>mid) change(2*now+1,x,y,mid+1,r,t);
    push_up(now);
}
double ask(int now,int x,int y,int l,int r){
    if(x<=l&&r<=y){
        return sum[now];
    }
    spread(now,l,r);
    int mid=(l+r)/2;
    double ans=0;
    if(x<=mid) ans+=ask(now*2,x,y,l,mid);
    if(y>mid) ans+=ask(now*2+1,x,y,mid+1,r);
    return ans;
}
double ask2(int now,int x,int y,int l,int r){
    if(x<=l&&r<=y){
        return sum2[now];
    }
    spread(now,l,r);
    int mid=(l+r)/2;
    double ans=0;
    if(x<=mid) ans+=ask2(now*2,x,y,l,mid);
    if(y>mid) ans+=ask2(now*2+1,x,y,mid+1,r);
    return ans;
}
```

2-4.典例：线段树优化 DP

```
将一个长度为 n 的序列分为 k 段，每一段的价值是不同数的个数，使得总价值最大
void push_up(int now){
    sum[now]=max(sum[2*now],sum[2*now+1]);
}
void build(int now,int l,int r,int lun){
```

```
    if(l==r){
       sum[now]=dp[l-1][lun-1];  //注意，是 l-1！！！即转移方程的前半段，从 lun-1 转移
        return;
    }
    sum[now]=0;
    tag[now]=0;
    int mid=(l+r)/2;
    build(now*2,l,mid,lun);
    build(now*2+1,mid+1,r,lun);
    push_up(now);
}
//与最大值线段树相同
int pre[35005];
int pos[35005];
signed main(){
  //  freopen("1.in","r",stdin);
    cin>>n>>k;
    for(int i=1;i<=n;i++){
        cin>>a[i];
        pre[i]=pos[a[i]];
        pos[a[i]]=i;//一个新数对颜色种类的影响，就是给它到上一个相同值的段，+1
    }
    for(int i=1;i<=k;i++){
        build(1,1,n,i);
        for(int j=1;j<=n;j++){
            change(1,pre[j]+1,j,1,n,1);
            dp[j][i]=ask(1,1,j,1,n);
        }
    }
    cout<<dp[n][k];

}
```

2-5.线段树动态开点、分裂与合并

```
#include <bits/stdc++.h>
#include <bits/extc++.h>
using namespace std;
using namespace __gnu_pbds;
using namespace __gnu_cxx;
const int maxn=2e5+5;
```

```
int nn;
class SegmentTree
{
public:
    long long a[200 * maxn], add[200 * maxn];
    int cnt=1,ls[200 * maxn],rs[200 * maxn],root[200*maxn];//root 编号与 cnt 无关!
    inline void push_up(int rt) //向上更新
    {
        a[rt] = a[ls[rt]] + a[rs[rt]];
    }
void build(int &now,int l,int r){
    now=++cnt;//连续分配
    if(l==r){
      scanf("%lld", a+now);
        return;
    }
    int mid=(l+r)/2;
    build(ls[now],l,mid);
    build(rs[now],mid+1,r);
}


    inline void push_down(int rt, int l,int r) //向下更新
    {
        if (add[rt] != 0)
        {
            add[ls[rt]] += add[rt];
            add[rs[rt]] += add[rt];
            int mid=l+r>>1;
            a[ls[rt]] += add[rt] * (mid-l+1);
            a[rs[rt]] += add[rt] * (r-mid);
            add[rt] = 0;
        }
    }
    void updata1(int x, int y, long long k, int l, int r, int &rt) //线段树区间为从 l 到 r，把区间
x 到 y 每个数+k
    {
        if(!rt)
        {
            rt=cnt;
            cnt++;
        }
        if (x <= l && y >= r)
        {
            a[rt] += (r - l + 1) * k;
```

```
        add[rt] += k;
        return;
    }
    push_down(rt, l,r);
    int mid = (l + r) / 2;
    if (x <= mid)
        updata1(x, y, k, l, mid, ls[rt]);
    if (y > mid)
        updata1(x, y, k, mid + 1, r, rs[rt]);
    push_up(rt);
}
long long query(int x, int y, int l, int r, int rt) //线段树区间为从 l 到 r，询问区间 x 到 y 的和
{
    if(!rt)
        return 0;
    if (x <= l && y >= r)
        return a[rt];
    push_down(rt, l, r);
    int mid = (l + r) / 2;
    long long ans = 0;
    if (x <= mid)
        ans += query(x, y, l, mid, ls[rt]);
    if (y > mid)
        ans += query(x, y, mid + 1, r, rs[rt]);
    return ans;
}
void merge(int &x,int &y,int l,int r) //将 y 合并到 x
{
    if(!x){
        x=y;
        return;
    }
    if(!y){
        return;
    }
    if(l==r)
    {
        a[x]+=a[y];
        return;
    }
    int mid=l+r>>1;
    merge(ls[x],ls[y],l,mid);
    merge(rs[x],rs[y],mid+1,r);
    push_up(x);
```

```
    }
    void split(int &p,int &q,int x,int y,int l,int r) //将 p 中 x 到 y 的区间分裂给 q[q 不一定是空树]
    {
        if(!p)
            return;
        if(x<=l&&r<=y)
        {
           merge(q,p,l,r);
            p=0;
            return;
        }
        if(!q)
        {
            q=cnt;
            cnt++;
        }
        int mid=l+r>>1;
        if(x<=mid)
            split(ls[p],ls[q],x,y,l,mid);
        if(y>mid)
            split(rs[p],rs[q],x,y,mid+1,r);
        push_up(p);
        push_up(q);
    }
}st;

int main()
{
   // freopen("1.in","r",stdin);
  //  freopen("1.out","w",stdout);
    int n,m,tmp,op,p,x,y,cntp=1;
    scanf("%d %d",&n,&m);
    for(int i=1;i<=n;i++)
    {
        scanf("%d",&tmp);
        st.updata1(i,i,tmp,1,n,st.root[1]);
    }
    while(m--)
    {
        scanf("%d",&op);
        switch (op)
        {
            case 0: 将可重集 p 中大于等于 x 且小于等于 y 的值移动到一个新的可重集中（或指定集合）
```

```
                scanf("%d %d %d",&p,&x,&y);
                cntp++;
                st.split(st.root[p],st.root[cntp],x,y,1,n);
                break;
            case 1: 将可重集 t 中的数放入可重集 p，且清空可重集 t
                scanf("%d %d",&p,&x);
                st.merge(st.root[p],st.root[x],1,n);
                break;
            case 2: 在 p 这个可重集中加入 x 个数字 q
                scanf("%d %d %d",&p,&x,&y);
                st.updata1(y,y,x,1,n,st.root[p]);
                break;
            case 3: 查询可重集 p 中大于等于 x 且小于等于 y 的值的个数
                scanf("%d %d %d",&p,&x,&y);
                printf("%lld\n",st.query(x,y,1,n,st.root[p]));
                break;
            case 4: 查询在 p 这个可重集中第 k 小的数，不存在时输出 -1
                scanf("%d %d",&p,&x);
                if(st.query(1,n,1,n,st.root[p])<x)
                    printf("-1\n");
                else
                    printf("%d\n",st.query2(x,1,n,st.root[p]));
                break;
        }
    }
}
```

3.可持久化线段树（主席树）

可持久化线段树保留每次“更改”或每次“加入”的状态，即可视为在时空上重合的，n 棵线段树。用于求第任意次修改后的树的状态，求指定区间内的第 k 大的数。

```
//求区间第 k 大数

#include<bits/stdc++.h>
using namespace std;
#define int long long

int n,m;
struct node{
    int v;
```

```
    int left;
    int right;
};
node node[20000005];
int val[20000005];
int tmp[20000005];
int root[20000005];
int cnt=0;
void push_up(int now){
    node[now].v=node[node[now].left].v+node[node[now].right].v;
}
void change(int &now,int l,int r,int q,int v=1){//动态开点
    node[++cnt]=node[now];//暂时连接原 node 节点下的节点(left 和 right)
    now=cnt;//操作新生成的点
    if(l==r){
        node[now].v+=1;
        return;
    }
    int mid=(l+r)>>1;
    if(q<=mid){
        change(node[now].left,l,mid,q,v);
    } else{
        change(node[now].right,mid+1,r,q,v);
    }
    push_up(now);
}
int query(int ll,int rr,int l,int r,int q){
    if(l==r){
        return l;
    }
    int mid=(l+r)>>1;
    int dec=node[node[rr].left].v-node[node[ll].left].v;
    if(dec>=q){//在左边
        return query(node[ll].left,node[rr].left,l,mid,q);
    }
    else{
        return query(node[ll].right,node[rr].right,mid+1,r,q-dec);
    }
}
signed main(){
    freopen("1.in","r",stdin);
    cin>>n>>m;
    for(int i=1;i<=n;i++){
        cin>>val[i];
```

```
        tmp[i]=val[i];
    }
    sort(tmp+1,tmp+1+n);
    int total=unique(tmp+1,tmp+1+n)-tmp-1;//总数记得减1！因为tmp从1开始存储
    for(int i=1;i<=n;i++){
        val[i]= lower_bound(tmp+1,tmp+1+total,val[i])-tmp;
        root[i]=root[i-1];//每一个点都相当于一个新的版本；差分性质，从上一个点继承
        change(root[i],1,total,val[i],1);
    }



    int ia,ib,k;
    for(int i=1;i<=m;i++){
        cin>>ia>>ib>>k;
        cout<<tmp[query(root[ia-1],root[ib],1,total,k)]<<endl;//映射原数
    }
}
```

3-1.历史版本查询

修改一个数组的值，并保留每一个历史版本。每一次查询也会多一个历史版本。

```
struct node{
    int v;
    int left;
    int right;
};
node node[256000005];
int val[256000005];
int root[256000005];
int cnt=0;

void build(int &now,int l,int r){
    now=++cnt;//连续分配
    if(l==r){
       node[now].v=val[l];
        return;
    }
    int mid=(l+r)/2;
    build(node[now].left,l,mid);
    build(node[now].right,mid+1,r);
}
```

```
void change(int &now,int l,int r,int num,int v){//动态开点
    node[++cnt]=node[now];//暂时连接原 node 节点下的节点(left 和 right)
    now=cnt;//操作新生成的点
    if(l==r){
        node[now].v=v;
        return;
    }
    int mid=(l+r)>>1;
    if(num<=mid){

        change(node[now].left,l,mid,num,v);
    } else{
        change(node[now].right,mid+1,r,num,v);
    }
}
int query(int now,int l,int r,int num){
    if(l==r){
        return node[now].v;
    }
    int mid=(l+r)>>1;
    if(num<=mid){//在左边
        return query(node[now].left,l,mid,num);
    }
    else{
        return query(node[now].right,mid+1,r,num);
    }
}
signed main(){
    //freopen("1.in","r",stdin);
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);
    cin>>n>>m;
    for(int i=1;i<=n;i++){
        cin>>val[i];
    }
    build(root[0],1,n);


    int ia,ib,opt,k;
    for(int i=1;i<=m;i++){
        cin>>ia>>opt;
        if(opt==1){
            cin>>ib>>k;
```

```
            root[i]=root[ia];
            change(root[i],1,n,ib,k);
        }else{
            cin>>ib;
            cout<<query(root[ia],1,n,ib)<<"\n";
            root[i]=root[ia];
        }
    }
}
```

4. 树状数组

树状数组是一种支持 单点修改 和 区间查询 的，代码量小的数据结构，常常用来处理前缀和的修改问题。普通树状数组维护的信息及运算要满足 **结合律** 和 **消去律**,如加法(和)、乘法（积）、异或等。

复杂度保证：我们总能将一段前缀 [1, n] 按 **lowerbit** 拆成 不多于 logn 段区间，使得这 logn 段区间的信息是已知的，从而实现合并。其中 a[k] 管辖着 lowerbit(k)个元素。

```
class BIT //树状数组
{
public:
    long long tree[maxn];
    int n;
    int lowbit(int x)
    {
        return x & (-x);
    }
    void add(int x, long long d) //将第 x 个数+k
    {
        while (x <= n)
        {
            tree[x] += d;
            x += lowbit(x);
        }
    }
    long long sum(int x) //求前 x 个数的和
    {
        long long ans = 0;
```

```
        while (x > 0)
        {
            ans += tree[x];
            x -= lowbit(x);
        }
        return ans;
    }
};
```

求 a[i]右边，比 a[i]大的个数：

```
for(int i=1;i<=n;i++){
    righ[i]=ran[i]-sum(ran[i])-1; //ran[i]为离散化后的，并且和原数据的大小关系相反
    add(ran[i],1);

}
```

4-1.差分树状数组

```
//实现区间修改，单点查询
void add(int l,int r,int x){

    for(int i=l;i<=n;i+=(i&-i)){
        tree[i]+=x;
    }
    for(int i=r+1;i<=n;i+=(i&-i)){
        tree[i]-=x;
    }
}
Void print(int i){
          Cout<<sum(i)+a[i]<<endl;
}
```

4-2.典例：扫描线

```
//给出一系列区间，给出一系列线段。扫描线求系列区间包含的线段条数和

for(int i=1;i<=m;i++){ //b 内是查询区间，按右端点从小到大排序；v 是线段，右端点非递减排序
    while(v[j].r<=b[i].r&&j<v.size()){
        add(v[j].l,1);//树状数组
        j++;
```

```
    }
    ans+=j-sum(b[i].l-1); //j 是处理过的线段条数
}
cout<<ans;
```

5. ST 表

ST 表（Sparse Table，稀疏表）是用于解决 **可重复贡献问题** 的数据结构。ST 表基于倍增 思想，可以做到 nlogn 预处理，O（1） 回答每个询问。但是不支持修改操作。

```
int a[100005];

int maxn[100005][25];

void init(){

    for(int j=1;(1<<j)<=n;j++){

        for(int i=1;i+(1<<j)-1<=n;i++){

            maxn[i][j]=max(maxn[i][j-1],maxn[i+(1<<(j-1))][j-1]);

        }

    }

}

int find(int ia,int ib){

    int k=(int)log2(ib-ia+1);

    return max(maxn[ia][k],maxn[ib-(1<<k)+1][k]);

}


for(int i=1;i<=n;i++){

       a[i]=read();

       maxn[i][0]=a[i];
```

```
    }

    init();
```

6. 优先队列

```
//优先队列运算符重载（从小到大排序）：
struct t{
    int val;
    bool operator <(const t&ia) const{  //注意，是重载小于号！！！！！
        return val>ia.val;
    }
};
priority_queue<t> q;

//动态求第 j 大的数的数
struct t{
    int val;
    bool operator < (const t&ia)const{
        return val>ia.val;
    }
};
priority_queue<t> q;
priority_queue<int> q2;
for(int j=1;j<=m;j++){//q2 为递减队列，q 为递增队列。q2 连接 q
        if(!q2.empty()&&a[j]<q2.top()){
            q.push({q2.top()});
            q2.pop();
            q2.push({a[j]});
        }
        else q.push({a[j]});
    }
    cout<<q.top().val<<endl;
    q2.push(q.top().val);
    q.pop();
}
```

二、字符串处理

1. 最长递增子序列 最大长度

1-1.单调数组解法

```
int choose_2(int ll,int rr){
  g[0]=-1e9;
  cnt=0;
  for(int i=ll;i<=rr;i++){
      if(a[i]>g[cnt]) g[++cnt]=a[i]; //g 数组单调递增
      else{  //查询从左往右第一个≥a[i]的位置
          int l=1;
          int r=cnt;
          int ans=0;
          while(l<=r){
              int mid=(l+r)/2;
              if(g[mid]>=a[i]){
                  r=mid-1;
                  ans=mid;
              }
              else{
                  l=mid+1;

              }
          }
          g[ans]=a[i];
      }
  }
  return cnt;//最长递增子序列最大长度，等价于不递增子序列的最少个数，g 值递增
};
```

1-2.树状数组解法

```
最长上升：
for (int i = 1; i <= n; i++) {
   int q = query(a[i] - 1);  // 找到[1,a[i]-1]中的最大值
   add(a[i], q + 1);  //这个最大值即是有效的转移 加入到树状数组中去
```

```
    res2 = max(res2, q + 1);
  }
最长不上升：
  for (int i = n; i >= 1; i--) {
    int q = query(a[i]);
    add(a[i], q + 1);
    res1 = max(res1, q + 1);
  }
```

1-3.DP 解法（N^2）

```
  for (int i = 1; i <= n; i++) {
    f[i] = 1;
    for (int j = 1; j < i; j++) {
      if (a[j] < a[i]) {  // j->i
        f[i] = max(f[i], f[j] + 1);
      }
    }
```

2. KMP

时间复杂度为 O（n+m）。暴力匹配在随机输入下，复杂度也为 O（n+m）。

```
class KMP   //此代码同时将答案映射到[1,s2.size()]上，s2 为小串
{
public:
    string s1, s2;
    int next[maxn];
    void getFail() //预处理 next 数组
    {
          // string s2=s[second]+" "+s[first]; //处理最长相等后前缀时使用
        next[0] = 0;
        next[1] = 0;
            num[0]=0;
            num[1]=1;
        for (int i = 1; i < s2.size(); i++)
        {
            int j = next[i];
            while (j != 0 && s2[i] != s2[j])
                j = next[j];//长度映射长度；j 即使 fail 后的长度，也是下一个字符的下标
```

```
            if (s2[i] == s2[j]) j++;
                next[i + 1] = j;
                    num[i+1]=num[j]+1;//所有公共前后缀计数(不同于 next 记录最大值长度)
        }

    }
    int find(int l,int r) //在 s1 的 l 到 r 范围内查找（s1 从 0 存储）
    {
        int j = 0;
        for (int i = l; i <= r; i++) //KMP 算法
        {
            while (j != 0 && s1[i] != s2[j])
                j = next[j];
            if (s1[i] == s2[j])
                j++;
            if (j == s2.size())
                return i - s2.size() + 1;
        }
        return -1;
    }
};
          int get(int i){  //求最短前后缀的长度
   if(tmp[i]) return tmp[i];

   if(inext[i]!=0&&inext[inext[i]]==0){
       return tmp[i]=inext[i];
   }
   return  tmp[i]=get(inext[i]);
}
```

3. 字典树

字典树用边来代表字母，而从根结点到树上某一结点的路径就代表了一个字符串。常用于计算大量字符串中，某个字符串从根节点开始，出现的次数。

```
class Trie
{
public:
    int trie[3000006][62]; //字典树主体，第一维为字符串可能的最大长度，第二维为可能出现的字符种类
```

```
    int num[3000006]; //字典树每个节点的计数器
    int p = 1;
    inline int reflect(char s) //字符映射
    {
        if (s >= '0' && s <= '9')
            return s - '0';
        else if (s >= 'A' && s <= 'Z')
            return s - 'A' + 10;
        else
            return s - 'a' + 36;
    }
    void insert(string &s) //将字符串 s 插入字典树
    {
        int now = 0, n;
        for (int i = 0; i < s.size(); i++)
        {
            n = reflect(s[i]);
            if (trie[now][n] == 0)
            {
                trie[now][n] = p;
                p++;
            }
            now = trie[now][n];
            num[now]++;
        }
          num[now]--;
          Finish[now]++;
    }
    int find(string &s) //查找字符串
    {
        int now = 0, n;
        for (int i = 0; i < s.size(); i++)
        {
            n = reflect(s[i]);
            if (trie[now][n] == 0)
                return 0;
            else
                now = trie[now][n];
        }
        return num[now];
    }
};
```

3-1.AC 自动机

AC（Aho–Corasick）自动机是 以 Trie 的结构为基础，结合 KMP 的思想 建立的自动机，用于解决多模式匹配等任务。给你一个文本串 S 和 n 个模式串 T，请你分别求出每个模式串 T 在 S 中出现的次数。

```
struct node{
    int a[30];
    int fail;
    int num;
    int ans;
}AC_[200005];

int cnt=0;
void build(int noww){
    int now=0;
    for(int i=0;i<s[noww].size();i++){
        if(AC_[now].a[s[noww][i]-'a']!=0){
            now=AC_[now].a[s[noww][i]-'a'];
        }
        else {
            AC_[now].a[s[noww][i] - 'a'] = ++cnt;
            now=AC_[now].a[s[noww][i]-'a'];
        }
    }
if(AC_[now].num==0){
        AC_[now].num=noww;
          Map[noww]=noww;
    }
    else Map[noww]=AC_[now].num; //这一步将所有相同的字符串的编号，映射到统一编号上

}
void get_fail(){  //核心函数：利用 bfs 构建失配函数
    AC_[0].fail=0;
    queue<int> q;
    for(int i=0;i<26;i++){
        if(AC_[0].a[i]!=0){
            AC_[AC_[0].a[i]].fail=0;
            q.push(AC_[0].a[i]);  //第一个字符压入队列中
```

```
        }
    }
    while(!q.empty()){
        int iq=q.front();
        q.pop();
        for(int i=0;i<26;i++){
            if(AC_[iq].a[i]!=0){
                AC_[AC_[iq].a[i]].fail=AC_[AC_[iq].fail].a[i];
                q.push(AC_[iq].a[i]);//压入新的有效点
           in[AC_[AC_[iq].fail].a[i]]++;//入度++

            }
            else{
                AC_[iq].a[i]=AC_[AC_[iq].fail].a[i];//特殊优化，减少匹配次数
            }
        }
    }
}
string ss;
void check(){
    int now=0;
    for(int i=0;i<ss.size();i++){
        now=AC_[now].a[ss[i]-'a'];
            AC_[now].ans++;
    }
    topu();
}
void topu(){
    queue<int> q;
    for(int i=1;i<=cnt;i++){
        if(in[i]==0) q.push(i);
    }
    while(!q.empty()){
        int iq=q.front();
        q.pop();

            jishu[AC_[iq].num]+=AC_[iq].ans; //按照 num 统计答案。Build 时建立了从字符串编号到 num
的映射

        AC_[AC_[iq].fail].ans+=AC_[iq].ans;
        in[AC_[iq].fail]--;
        if(in[AC_[iq].fail]==0) q.push(AC_[iq].fail);
    }
}
```

4. 01 字典树

```
//在树中找两个结点，求异或值最大的路径
//以下处理树上两点间最大的异或和
int sum[100005];
int id;
int tree[2000005][2];
void dfs(int now,int fa){  //记录从根开始的异或和
    for(int i=0;i<v[now].size();i++){
        if(v[now][i].to==fa) continue;
        sum[v[now][i].to]=sum[now]^v[now][i].val;
        dfs(v[now][i].to,now);
    }
}
void build(){
    for(int j=1;j<=n;j++){
       int d=sum[j];
        int now=0;
        for(int i=(1<<MAXN);i>0;i>>=1){ //统一异或的高度，因为0也能改变异或值
            bool c=d&i;
            if(tree[now][c]){
                now=tree[now][c];
            }
            else{
                tree[now][c]=++id;
                now=tree[now][c];
                }
            }
    }

}
int query(){ //每次固定一个sum[j]，然后找和它异或和最大的路径
    int ans=0;
    for(int j=1;j<=n;j++){
        int d=sum[j];
        int tmp=0;
        int now=0;
        for(int i=(1<<MAXN);i>0;i>>=1){
            bool c=d&i;
            if(tree[now][!c]){
                tmp+=i; //求和
                now=tree[now][!c];
            }
```

```
            else now=tree[now][c];
        }
        ans=max(ans,tmp);
    }
    return ans;

}
```

4-1.可持久化 01 字典树

```
//在 l 到 r 区间内，找一个数，其与 val 异或能得到的最大值
int id=0;
int tree[30000005][2];
int num[30000005];
int root[200005];

bitset<MAXN> s;
void insert(int last,int &new_root,int val){
    s=val;//二进制分解
    if(!new_root){
        new_root=++id;//创建新的历史版本
    }
    int now=new_root;
    for(int i=MAXN-1;i>=0;i--){ //统一异或的高度，因为 0 也能改变异或值
       if(s[i]){
           tree[now][0]=tree[last][0];//相同：沿用旧值
           tree[now][1]=++id;//不同：使用新节点
           num[id]=num[tree[last][1]]+1;//记录这个差异
           now=id;
           last=tree[last][1];
       }else{
           tree[now][1]=tree[last][1];
           tree[now][0]=++id;//新节点
           num[id]=num[tree[last][0]]+1;
           now=id;
           last=tree[last][0];
       }
    }
}
bitset<MAXN> b;
int find(int ll,int rr,int val){
    s=val;//二进制分解
```

```
    for(int i=MAXN-1;i>=0;i--){ //统一异或的高度，因为0也能改变异或值
        if(s[i]){
            if(num[tree[rr][0]]-num[tree[ll][0]]!=0){
                b[i]=true;//0异或1得到1
                ll=tree[ll][0];
                rr=tree[rr][0];
            }else{
                b[i]=false;
                ll=tree[ll][1];
                rr=tree[rr][1];
            }
        }else{
            if(num[tree[rr][1]]-num[tree[ll][1]]!=0){
                b[i]=true;1异或0得到1
                ll=tree[ll][1];
                rr=tree[rr][1];
            }else{
                b[i]=false;
                ll=tree[ll][0];
                rr=tree[rr][0];
            }
        }
    }
    return b.to_ulong();
}
```

4-2.典例：可乐

```
 //给出数组a，要求a[i]异或x≤k，构造一个x，使得满足条件的a[i]数目最多
void f(){
    int sum=0,kk=k;
    int len1=0,len2=0;
    for(int i=1;i<50;i++) s[i]=0,tmp[i]=0;
    while(dd){
        s[++len1]=dd%2;
        dd>>=1;
    }
    while(kk){
        tmp[++len2]=kk%2;
        kk>>=1;
    }
    int len=max(len1,len2);
```

```
    for(int i=1;i<=len/2;i++){
        swap(s[i],s[len-i+1]);
        swap(tmp[i],tmp[len-i+1]);
    }  //倒序存储：左大右小


    for(int i=1;i<=len;i++){
        if(tmp[i]==0){
            sum=sum*2+s[i];  //如果为0，只能与a[i]相同
        }
        else{
            flag[(sum*2+s[i])*(1<<(len-i))]++;
            flag[(sum*2+1+s[i])*(1<<(len-i))]--;  如果为1，且与a[i]相同时，后面位可任意
            sum=sum*2+(s[i]^1); 如果为1，且与a[i]不同时，继续处理
        }
    }
    flag[sum]++;
    flag[sum+1]--;  //差分操作
}
```

5.马拉车求回文串

```
void proc(){
    a.push_back('*');
    for(int i=0;i<ss.size();i++){
        a.push_back('-');
        a.push_back(ss[i]);
    }
    a.push_back('-');
    a.push_back('+');   //先处理字符串，首尾防止越界  aba---》 *-a-b-a-+
}
int manacher(){  //复杂度为O（n）
    int righ=0,mid=0;
    int ans=0;
    for(int i=2;i<a.size()-2;i++){
        if(i<righ){
            len[i]=min(righ-i,len[2*mid-i]);  //利用mid至righ这一目前包含i的最长回文串 的 对称性质，来加速运算
        }else{
            len[i]=1;
        }
        while(a[i+len[i]]==a[i-len[i]]) len[i]++;//暴力扩展
        if(i+len[i]>righ){
```

```
            righ=i+len[i];  //更新区间，使新的区间更可能包含要计算的区间
            mid=i;
            ans=max(ans,len[i]);
        }

    }
    return ans-1;
}
```

6. k 进制哈夫曼树

```
struct node{
    int val;
    int high;
    bool operator <(const node&ia) const{
        if(val==ia.val){
            return high>ia.high;
        }
        return val>ia.val;
    }
};




priority_queue<node> q;
while((q.size()-1)%(k-1)!=0) q.push({0,1});
while(q.size()!=1){
    int sum=0;
    node tmp;
    for(int i=1;i<=k&&!q.empty();i++){
        node iq=q.top();
        q.pop();
        sum+=iq.val;
        maxn=max(maxn,iq.high); //最大路径长度
        ans+=iq.val;//最小 WPL
    }
    tmp.val=sum;
    tmp.high=maxn+1;
    q.push(tmp);
}
cout<<ans<<endl<<maxn;
```

三、图论

1.正边权最短路，Dijkstra（贪心）

复杂度：O（（n+m）logm）

将结点分成两个集合：已确定最短路长度的点集（记为 S 集合）的和未确定最短路长度的点集（记为 T 集合）。一开始所有的点都属于 T 集合。

然后重复这些操作：

1.从 T 集合中，选取一个最短路长度最小的结点，移到 S 集合中。

2.对那些刚刚被加入 S 集合的结点的所有出边执行松弛操作。

直到 T 集合为空，算法结束。

```
struct edge{
    int to;
    int w;
};
struct node{
    int num;
    int dis;
    bool operator<(const node& that) const
    {
        return dis > that.dis;  //重载后小的数在前
    }

};
vector<edge> v[2*1000000+5];
int dis[2*1000000+5];
void dij(int s){
    priority_queue<node> q;
    for (int i = 1; i <= n; i++) //初始化
        if (i != s)
            dis[i] = 2147483647;
    q.push({ s,0 });
    while(!q.empty()) {
        node iq=q.top();
        q.pop();
        if(dis[iq.num]!=iq.dis) continue;
```

```
        for(int i=0;i<v[iq.num].size();i++){
           if(dis[v[iq.num][i].to]==dis[iq.num]+v[iq.num][i].w){
                ans[v[iq.num][i].to]=ans[v[iq.num][i].to]+ans[iq.num];
            }//最短路计数
            if(dis[v[iq.num][i].to]>dis[iq.num]+v[iq.num][i].w){
                ans[v[iq.num][i].to]=ans[iq.num]; //最短路计数
                dis[v[iq.num][i].to]=dis[iq.num]+v[iq.num][i].w;
           q.push({v[iq.num][i].to,dis[v[iq.num][i].to]});

            }


        }
    }
}
```

1-1.分层图

```
//（从 A 到 B，但可以做 k 次“飞机”：没有边权值）：

 while(m--){

        scanf("%d %d %d",&ia,&ib,&ic);

        v[ia].push_back({ib,ic});

        v[ib].push_back({ia,ic});

        for(int i=1;i<=k;i++){

            v[ia+i*n].push_back({ib+i*n,ic});

            v[ib+i*n].push_back({ia+i*n,ic});

            v[ia+(i-1)*n].push_back({ib+i*n,0});

            v[ib+(i-1)*n].push_back({ia+i*n,0});

        }

    }

    for(int i=1;i<=k;i++){
```

```
        v[t+(i-1)*n].push_back({t+i*n,0});  //未必需要坐 k 次飞机

    }

    dij(s);

        printf("%d",dis[t+k*n]);
```

1-2.典例：佳佳的魔法药水

```
//边权是到某个点的最短路。1 份 A 药水混合 1 份 B 药水就可以得到 1 份 C 药水。最少花多少钱可以配制成功这种珍贵的药水；共有多少种不同的花费最少的方案。
void dij(){
    while(!q.empty()) {
        node iq=q.top();
        q.pop();
        if(dis[iq.num]!=iq.dis) continue;
        flag[iq.num]=1;
        for(int i=0;i<v[iq.num].size();i++){
            if(flag[v[iq.num][i].w]) {
                if (dis[v[iq.num][i].to] > dis[iq.num] + dis[v[iq.num][i].w]) {
                    dis[v[iq.num][i].to] = dis[iq.num] + dis[v[iq.num][i].w];
                    ans[v[iq.num][i].to] = ans[iq.num]* ans[v[iq.num][i].w];
                    q.push({v[iq.num][i].to, dis[v[iq.num][i].to]});
                }
                else  if (dis[v[iq.num][i].to] == dis[iq.num] + dis[v[iq.num][i].w]){
                    ans[v[iq.num][i].to]+= ans[iq.num]* ans[v[iq.num][i].w];
                }
            }


        }
    }
}
调用：
for(int i=0;i<n;i++){
    scanf("%lld",&dis[i]);  //直接买的价格
    q.push({i,dis[i]});
    ans[i]=1; //默认有一种方案
}
while(scanf("%lld %lld %lld",&ia,&ib,&ic)!=EOF){
    v[ia].push_back({ic,ib});  //ia+ib==ic
```

```
    v[ib].push_back({ic,ia});
}
```

1-3.典例：同余最短路

```
//总高度是 h，有一个电梯，可以向上移动 x 层，向上移动 y 层，向上移动 z 层，或回到第一层。问一共能到达多少层。
for(int i=0;i<x;i++){
    v[i].push_back({(i+y)%x,y});
    v[i].push_back({(i+z)%x,z});
}
dij(0);
int ans=0;
for(int i=0;i<x;i++){
    if(h>=dis[i]) ans+=(h-dis[i])/x+1;
}
```

2. 含负边权最短路，SPFA（贪心）

一条最短路中，最多含有 n 个节点。而一个节点每一次入队，就会为当前的队列内的最短路的长度增加 1

```
class edge
 {
 public:
     int to, val;
 };
 int n;
 vector<edge> G[maxn];
 int dis[maxn],neg[maxn];
 bool flag[maxn];
 bool SPFA(int s) //s 为源点，若图中有负环则返回 false
 {
     queue<int> q;
     for (int i = 1; i <= n; i++)
         dis[i] = 2147483647;
     q.push(s);
           dis[s]=0;
           flag[s] = true;
           neg[s]++;
     int now;
```

```
    while (!q.empty())
    {
        now = q.front();
        q.pop();
        flag[now] = false;
        for (auto i:G[now])
        {
            if (dis[i.to] > dis[now] + i.val)
            {
                dis[i.to] = dis[now] + i.val;
                if (!flag[i.to])
                {
                    flag[i.to] = true;
                    q.push(i.to);
                    neg[i.to]++;
                    if(neg[i.to]>=n)
                        return false;
                }
            }
        }
    }
    return true;
}
```

2-1.典例：差分约束

```
//已知不等式组，求任意一组满足这个不等式组的解。
for(int i=1;i<=n;i++){
    dis[i]=2147483647;
    v[0].push_back({i,0});
}
for(int i=1;i<=m;i++){
    scanf("%d %d %d",&ia,&ib,&ic);
    v[ib].push_back({ia,ic});   //ia-ib <=ic 时，转为 ia （汇）<=ib（源）+ic

}
        dis[0]=0;
        queue<int> q;
        q.push(0);
        flag[0]=true;
        ans[0]++;
        while(!q.empty()){
```

```
            int id=q.front();
            q.pop();
            flag[id]=false;
            for(int i=0;i<v[id].size();i++){
                if(dis[v[id][i].to]>dis[id]+v[id][i].val){
                    dis[v[id][i].to]=dis[id]+v[id][i].val;
                    if(flag[v[id][i].to]==false){
                        flag[v[id][i].to]=true;
                        ans[v[id][i].to]++;
                        if(ans[v[id][i].to]>=n+1){
                            printf("NO\n");
                           return 0;
                        }
                        q.push(v[id][i].to);
                    }
                }
            }
        }
        for(int i=1;i<=n;i+=1){
            printf("%d ",dis[i]);
        }
```

3.含负边权最短路，Floyd （dp）

如果存在一条通过节点 k 的路径，使得从 i 到 j 的距离更短，那么就更新这个距离。

```
for(int i=1;i<=n;i++){
    for(int j=1;j<=n;j++){
        if(i!=j) dis[i][j]=2147483647;
    }
}
for(int k=1;k<=n;k++){
    for(int i=1;i<=n;i++){
        if(k!=i){
            for(int j=1;j<=n;j++){
                if(j!=i&&j!=k){
                    if(dis[i][j]>dis[i][k]+dis[k][j]){
                        dis[i][j]=dis[i][k]+dis[k][j];
                    }
                }
```

```
            }
        }
    }
}
```

3-1.典例：传送门

```
//在 l 和 r 之间建立了传送门，0 代价。求更改后的最短路。注意，只能改一条边！！！
for (int l = 1; l <= n; l++) {
    for (int r = l + 1; r <= n; r++) {
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                tmp[i][j] = a[i][j];
            }
        }
        tmp[l][r]=tmp[r][l]=0;
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                if (tmp[i][l] != 2147483647 && tmp[l][j] != 2147483647 && tmp[i][j] > tmp[i][l]
+tmp[l][j]) {
                    tmp[i][j] = tmp[i][l] + tmp[l][j];
                }
            }
        }
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                if (tmp[i][r] != 2147483647 && tmp[r][j] != 2147483647 && tmp[i][j] > tmp[i][r] +
tmp[r][j]) {
                    tmp[i][j] = tmp[i][r] + tmp[r][j];
                }
            }
        }
    }
}
```

4.最小生成树， Kruskal（贪心）

从最小边权的边开始，按边权从小到大依次加入，如果某次加边产生了环，就扔掉这条边，直到加入了 n-1 条边，即形成了一棵树。

```
int n, m;
```

```
class edge
{
public:
    int from, to, val;
    bool operator<(edge& that)
    {
        return val < that.val;
    }
}e[maxe];
int s[maxn];
int find(int x) //并查集查找
{
    if (s[x] == x)
        return x;
    else
    {
        s[x] = find(s[x]);
        return s[x];
    }
}
void union_set(int x, int y) //并查集合并
{
    x = find(x);
    y = find(y);
    if (x != y)
        s[y] = s[x];
}
void k() {
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        scanf("%lld %lld %lld", &bian[i].from, &bian[i].to, &bian[i].val);
    }
    sort(bian + 1, bian + m + 1, cmp);
    int count = 0;
    for (int i = 1; i <= m; i++) {
        if(bian[i].from==bian[i-1].from&&bian[i].to==bian[i-1].to){ flag[i]=1;continue;}
          //去除自环和重边，虽然没什么用
        int j = find(bian[i].from);
        int k = find(bian[i].to);
        if (j == k) continue;
        flag[i]=1;
        join(j, k);
        a[bian[i].from].push_back({bian[i].to,bian[i].val});
        a[bian[i].to].push_back({bian[i].from,bian[i].val});
```

```
        total += bian[i].val;
        count++;
        if (count == n - 1) return;
    }
    printf("orz");
    exit(0);
}
```

4-1.次小生成树

```
#include<bits/stdc++.h>
using namespace std;
#define int long long
struct edge {
    int from;
    int to;
    int val;
}bian[2000005];
struct node{
    int to;
    int val;
};

int father[100005];
int flag[300005];
bool cmp(edge& a, edge& b) {
    return a.val < b.val;
}
int find(int x) {
    if (father[x] <=0 ) return x;
    else return father[x] = find(father[x]);
}
bool join(int a, int b) {
    int ia = find(a);
    int ib = find(b);
    if (ia == ib) return true;
    if (father[ia] > father[ib]) father[ia] = ib;
    else if (father[ia] < father[ib]) father[ib] = ia;
    else {
        father[ib] = ia;
        father[ia]--;
    }
    return true;
```

```
}
int total;
int n, m;
vector<node> a[100005];
void k() {
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        scanf("%lld %lld %lld", &bian[i].from, &bian[i].to, &bian[i].val);
    }
    sort(bian + 1, bian + m + 1, cmp);
    int count = 0;
    for (int i = 1; i <= m; i++) {
        if(bian[i].from==bian[i-1].from&&bian[i].to==bian[i-1].to){ flag[i]=1;continue;}
        int j = find(bian[i].from);
        int k = find(bian[i].to);
        if (j == k) continue;
        flag[i]=1;
        join(j, k);
        a[bian[i].from].push_back({bian[i].to,bian[i].val});
        a[bian[i].to].push_back({bian[i].from,bian[i].val});
        total += bian[i].val;
        count++;
        if (count == n - 1) return;
    }
    printf("orz");
    exit(0);
}
int ceng[500005];
int fa[500005][25];
int maxn[500005][25];
int maxn2[500005][25];
void dfs(int now,int faa,int val){
    ceng[now]=ceng[faa]+1;
    fa[now][0]=faa;
    maxn[now][0]=val;
    maxn2[now][0]=0;
    for(int i=1;(1<<i)<=ceng[now];i++){
        fa[now][i]=fa[fa[now][i-1]][i-1];
        maxn[now][i]=max(maxn[now][i-1],maxn[fa[now][i-1]][i-1]);
        maxn2[now][i]=max(maxn2[now][i-1],maxn2[fa[now][i-1]][i-1]);
        if(maxn[now][i-1]!=maxn[fa[now][i-1]][i-1]){
            maxn2[now][i]=max(min(maxn[fa[now][i-1]][i-1],maxn[now][i-1]),maxn2[now][i]);
        }
    }
```

```
    for(int i=0;i<a[now].size();i++){
        if(a[now][i].to==faa) continue;
        dfs(a[now][i].to,now,a[now][i].val);
    }
}
int LCA(int ia,int ib,int val){
    int tmp=0;
    if(ceng[ia]>ceng[ib]){
        swap(ia,ib);
    }
    for(int i=24;i>=0;i--){
        if(ceng[ia]==ceng[ib]) continue;
        if(ceng[ib]-(1<<i)>=ceng[ia]){
            tmp=max(tmp,(maxn[ib][i]==val?maxn2[ib][i]:maxn[ib][i]));
            ib=fa[ib][i];
        }
    }
    if(ia==ib) return tmp;
    for(int i=24;i>=0;i--){
        if(fa[ia][i]==fa[ib][i]) continue;
        else{
            if(max(maxn[ia][i],maxn[ib][i])!=val){
                tmp=max(tmp,max(maxn[ia][i],maxn[ib][i]));
            }
            else{
                int tmp1=min(maxn[ia][i],maxn[ib][i]);

tmp=max(tmp,tmp1==val?max(maxn2[ia][i],maxn2[ib][i]):max(tmp1,max(maxn2[ia][i],maxn2[ib][i])));
            }
            ia=fa[ia][i],ib=fa[ib][i];
        }
    }
    return max(maxn[ib][0]==val?0:maxn[ib][0],max(tmp,maxn[ia][0]==val?0:maxn[ia][0]));
}

signed main()
{
//    freopen("1.in","r",stdin);
    k();
    int root=-1;
    for(int i=1;i<=m;i++){
        if(flag[i]==1){
            root=bian[i].from;
            break;
```

```
        }
    }
    dfs(root,0,0);
    int ans=1e18;
    for(int i=1;i<=m;i++){
        if(flag[i]==1) continue;
        if(bian[i].from==bian[i].to) continue;
        int val=LCA(bian[i].from,bian[i].to,bian[i].val);
        if(bian[i].val>val)  ans=min(ans,total-val+bian[i].val);
    }
    cout<<ans;
}
```

5. 树链剖分

将一棵树，分成诺干不重复的链，放入一个数组中。同时，属于同一棵子树的点，在数组中保持连续（**dfn 序**保证）。同时，将每个点的最大的儿子作为其重儿子，在向下经过一条 **轻边** 时，所在子树的大小至少会除以二。因此，树上的每条路径都可以被拆分成不超过 O(log n) 条重链。

```
int sum[400005];
int tag[400005];
int a[100005];
int wt[100005];
void push_up(int now){
    sum[now]=(sum[2*now]+sum[2*now+1])%p;
}
void build(int now,int l,int r){
    if(l==r){
        sum[now]=wt[l];
        return;
    }
    int mid=(l+r)/2;
    build(now*2,l,mid);
    build(now*2+1,mid+1,r);
    push_up(now);
}
void spread(int now,int l,int r){
```

```
    if(tag[now]){
        tag[now*2]=(tag[now*2]+tag[now])%p;
        tag[now*2+1]=(tag[now*2+1]+tag[now])%p;
        int mid=(l+r)/2;
        sum[now*2]=(sum[now*2]+tag[now]*(mid-l+1))%p;
        sum[now*2+1]=(sum[now*2+1]+tag[now]*(r-mid))%p;
        tag[now]=0;
    }
}
void change(int now,int x,int y,int l,int r,int t){
    if(x<=l&&r<=y){
        sum[now]=(sum[now]+t*(r-l+1))%p;
        tag[now]=(tag[now]+t)%p;
        return;
    }
    spread(now,l,r);
    int mid=(l+r)/2;
    if(x<=mid) change(2*now,x,y,l,mid,t);
    if(y>mid) change(2*now+1,x,y,mid+1,r,t);
    push_up(now);
}
int ask(int now,int x,int y,int l,int r){
    if(x<=l&&r<=y){
        return sum[now]%p;
    }
    spread(now,l,r);
    int mid=(l+r)/2;
    int ans=0;
    if(x<=mid) ans+=ask(now*2,x,y,l,mid);
    ans%=p;
    if(y>mid) ans+=ask(now*2+1,x,y,mid+1,r);
    return ans%p;
}

int ia,ib,ic;
vector<int> v[100005];
int dep[100005];
int fa[100005];
int siz[100005];
int son[100005];
int id[100005];

int top[100005];
int cnt=0;
```

```
void dfs1(int now,int father){
    dep[now]=dep[father]+1;
    fa[now]=father;
    siz[now]=1;
    int Maxson=-1;
    for(int i=0;i<v[now].size();i++){
        if(v[now][i]==father) continue;
            a[v[now][i].to]=v[now][i].val;    //点边转换：边权给点权
        dfs1(v[now][i],now);
        siz[now]+=siz[v[now][i]];
        if(siz[v[now][i]]>Maxson){
            Maxson=siz[v[now][i]];
            son[now]=v[now][i];
        }
    }
}
void dfs2(int now,int father){
    id[now]=++cnt;
    wt[cnt]=a[now];
numm[cnt]=now; //有时候题目要求输出特定点的编号，此处为了记录逆转换关系（和 id 互逆）
    top[now]=father;
    if(!son[now]) return;
    dfs2(son[now],father);
    for(int i=0;i<v[now].size();i++){
        if(v[now][i]==fa[now]||v[now][i]==son[now]) continue;
        dfs2(v[now][i], v[now][i]);
    }
}
void upRange(int l,int r,int t){
    while(top[l]!=top[r]){
        if(dep[top[l]]<dep[top[r]]) swap(l,r);
        change(1,id[top[l]],id[l],1,n,t);
        l=fa[top[l]];
    }
    if(dep[l]>dep[r]) swap(l,r);
    change(1,id[l],id[r],1,n,t);
 change(1,id[l],id[l],1,n,-t); //点边转换：去掉顶点的值

}
int qRange(int l,int r){
    int ans=0;
    while(top[l]!=top[r]){
        if(dep[top[l]]<dep[top[r]]) swap(l,r);
        ans+=ask(1,id[top[l]],id[l],1,n);//从根到下， id 单增
```

```
        ans%=p;
        l=fa[top[l]];
    }
    if(dep[l]>dep[r]) swap(l,r);
    ans+=ask(1,id[l],id[r],1,n);
            ans-=ask(1,id[l],id[l],1,n);//点边转换：去掉顶点的值
    return ans%p;
}
void upSon(int now,int t){
    change(1,id[now],id[now]+siz[now]-1,1,n,t);
 change(1,id[now],id[now],1,n,-t); //点边转换

}
int qSon(int now){
    return  ask(1,id[now],id[now]+siz[now]-1,1,n)%p-ask(1,id[now],id[now],1,n)%p;
//后半段是点边转换：去掉顶点的值
}



调用：
dfs1(R,0);
dfs2(R,R);
build(1,1,n);
```

6. 最近公共祖先 LCA

倍增算法的预处理时间复杂度为 nlogn，单次查询时间复杂度为 logn。

```
int ceng[500005];
vector<int> a[500005];
int fa[500005][25];
void dfs(int now,int faa){
    ceng[now]=ceng[faa]+1;
    fa[now][0]=faa;
    for(int i=1;(1<<i)<=ceng[now];i++){
        fa[now][i]=fa[fa[now][i-1]][i-1];
    }
    for(int i=0;i<a[now].size();i++){
        if(a[now][i]==faa) continue;
        dfs(a[now][i],now);
    }
}
```

```
int LCA(int ia,int ib){
     if(ia==ib) return ia; //本身就是一个点
    if(ceng[ia]>ceng[ib]){
        swap(ia,ib);
    }
    for(int i=24;i>=0;i--){
        if(ceng[ia]==ceng[ib]) continue;
        if(ceng[ib]-(1<<i)>=ceng[ia]){
            ib=fa[ib][i];
        }
    }
    if(ia==ib) return ia; //此时不 return，下面就犯错!
    if(ceng[ia]==0) return -1; //两个点不在一个树上!
    for(int i=24;i>=0;i--){
        if(fa[ia][i]==fa[ib][i]) continue;
        else{
            ia=fa[ia][i],ib=fa[ib][i];
        }
    }
    if(ceng[ia]==0) return -1; //两个点不在一个树上!
    return fa[ia][0];
}
```

6-1 图上 LCA-跑路机

```
for(int i=1;i<=m;i++){   //从点 1 到点 n，当距离为 2 次幂时，能在 1s 内到达。最短需要多少秒?
    cin>>ia>>ib;
    a[ia][ib][0]=1;
    dis[ia][ib]=1;
}
for(int i=1;i<64;i++){
    for(int j=1;j<=n;j++){
        for(int k=1;k<=n;k++){
            for(int s=1;s<=n;s++) {
                if (a[j][k][i-1]==1&&a[k][s][i-1]==1){
                    a[j][s][i]=1;
                    dis[j][s]=1;
                }
            }
        }
    }
}
```

```
for(int t=1;t<=n;t++){
    for(int i=1;i<=n;i++){
        for(int j=1;j<=n;j++){
            if(i!=j&&i!=t&&j!=t&&dis[i][t]!=1e18&&dis[t][j]!=1e18){
                dis[i][j]=min( dis[i][j],dis[i][t]+dis[t][j]);
            }
        }
    }
}
cout<<dis[1][n];
```

7. 树的直径

在一棵树上，从任意节点 y 开始进行一次 DFS，到达的距离其最远的节点 z 必为直径的一端。

```
int nod=0;
int ceng[100005];
void dfs(int now,int fa){
          Path[now]=fa;//记录路径
    ceng[now]=ceng[fa]+1;
    if(ceng[now]>ceng[nod]){
        nod=now;
    }
    for(int i=0;i<xintu[now].size();i++){
        if(tu[now][i]==fa) continue;
        dfs(tu[now][i],now);
    }
}
调用：
dfs(1,0);
ceng[nod]=0;//设置为新的根节点
dfs(nod,0);
for(int i=nod;i;i=path[i]){
    v2.push_back(i);
}
```

7-1.树的重心

重心性质：

1. 所有子树的大小小于总大小的一半
2. **所有点到重心的距离和最小**
3. 两棵树合并后，重心在两棵树重心的连线上

```
void zhongxing(int now,int fa){
    siz[now]=1;
    f[now]=0;
    for(int i=0;i<v[now].size();i++){
       if(v[now][i]==fa) continue;
       zhongxing(v[now][i],now);
       siz[now]+=siz[v[now][i]];
       f[now]=max(f[now],siz[v[now][i]]);
    }
    f[now]=max(f[now],n-siz[v[now]]);
    if(f[now]<f[center]){
        center=now;
    }
}
```

8. 拓扑排序

在 AOV 网中，顶点表示活动，弧表示活动间的优先关系。AOV 网中不应该出现环，这样就能够找到一个顶点序列，使得每个顶点代表的活动的前驱活动都排在该顶点的前面，这样的序列称为拓扑序列（一个 AOV 网的拓扑序列不是唯一的），由 AOV 网构造拓扑序列的过程称为拓扑排序。

```
 vector<int> G[maxn];
int rudu[maxn];
bool flag[maxn];
int ans[maxn];
int top = 0;
```

```
int n;
void tuopu()
{
    std::priority_queue<int,vector<int>,greater<int>> q; //若不需要按照字典序输出，可将优先队列换成普通队列
    for (int i = 0; i < n; i++)
    {
        if (rudu[i] == 0)
            q.push(i);
    }
    int now;
    while (!q.empty())
    {
        now = q.top();
        ans[top] = now;
        flag[now] = true;
        top++;
        q.pop();
        for (int i = 0; i < G[now].size(); i++)
        {
            rudu[G[now][i]]--;
            if (rudu[G[now][i]] == 0)
                q.push(G[now][i]);
        }
    }
}
```

9. 割点割边，tarjan

9-1. 无向图-割点

如果 y 是 x 的子节点且 low(y)≥dfn(x)，那么 x 就是割点。

由定义，y 在不经过 (x,y) 的情况下只能到达比 x 更晚访问到的节点，所以删去 (x,y) 后，y 必定与比 x 更早访问到的点不相连，就必然会分裂成一张不联通的子图。

```
map<int,int> ans;
stack<int> s;
void tarjan(int now,int fa) {
    int lstnum=0;
```

```
    low[now]=dfn[now] = ++cnt;
    s.push(now);
    for (int i = 0; i < a[now].size(); i++) {
        if(a[now][i]==fa){
            lstnum++;
            if(lstnum<2){
                continue;
            }
        }//如果有重边：即根到 B 有两条边，那么根和 B 为同一个连通块
        int child = a[now][i];
        if(dfn[child]){  //触底反弹
            low[now]=min(low[now],dfn[child]);
        }
        else {
            tarjan(child,now);
            if (dfn[now]<=low[child]) {
                ans[now]++;  //加入树根/割点
                while (s.top() != child) {
                    ans[s.top()]++; //统计圆方树的度数：度数＞2 的为割点。否则，根会被误判为割点（因为满足 low(y)≥dfn(root)）
                    s.pop();
                }
                ans[s.top()]++;
                s.pop();
            }
            low[now] = min(low[now], low[child]);
        }

    }
}

    调用：

for(int i=1;i<=n;i++){
    if(dfn[i]==0) tarjan(i,0);
}
```

9-2.无向图-割边

```
```

```
void tarjan(int now,int father){
    dfn[now]=cnt++;
    low[now]=dfn[now];
     int lstnum=0;
    for(int i=0;i<a[now].size();i++) {
        int child = a[now][i];

     if(child==father){
    lstnum++;
    if(lstnum<2)
        continue;
}//如果有重边：即 A 到 B 有两条边，那么 A 和 B 为同一个连通块
        if(child==father) continue;
        if (dfn[child]) {
            low[now]=min(low[now],dfn[child]);
        }
        else {
            tarjan(child, now);
            if (dfn[now] < low[child]) {          //处理割边，比如记录下来
                  v.push_back({child, now});
            }
            low[now]=min(low[now],low[child]);
        }

    }
}调用：for(int i=1;i<=n;i++){
    if(dfn[i]==0) tarjan(i,0);
}
```

9-3.无向图-边双连通分量

无向图的**缩点**问题，连通分量通过割边连接。一个边双联通分量中，断开任何一条边，都不能破坏分量内部的连通性。
一个点可能属于多个点双，但是一条边属于恰好一个点双。

```
void taijan(int now,int father){
    s.push(now);
    dfn[now]=cnt++;
    low[now]=dfn[now];
    int lstnum=0;
    for(int i=0;i<a[now].size();i++) {
        int child = a[now][i];
```

```
        if(child==father){
            lstnum++;
            if(lstnum<2)
                continue;
        }//如果有重边：即 A 到 B 有两条边，那么 A 和 B 为同一个连通块
        if (dfn[child]) {
            low[now]=min(low[now],dfn[child]);
        }
        else {
            taijan(child, now);
            low[now]=min(low[now],low[child]);
            }
        }

    if (dfn[now] == low[now]) {   //放在外面！！！
        while(s.top()!=now){
            id[s.top()]=cnt2;
            s.pop();
        }
        id[s.top()]=cnt2;
        s.pop();
        cnt2++;
    }

}
void suodian(){
    for(int i=1;i<=n;i++){
        if(dfn[i]==0) taijan(i,0);
    }
}

  void jiantu() {

    for (int i = 1; i <= n; i++) {

        for (int j = 0; j < a[i].size();j++){

            if(id[i]!=id[a[i][j]]){

                xintu[id[i]].push_back(id[a[i][j]]);

                xintu[id[a[i][j]]].push_back(id[i]);

            }
```

```
        }

    }

}
```

9-4. 无向图-点双连通分量、圆方树

圆方树：圆方交替、度数超过 2 的圆点，是割点、方点所连的点，属于同一个点双联通分量。

每个点双形成一个「菊花图」，多个「菊花图」通过原图中的割点连接在一起（因为点双的分隔点是割点）。

```
void tarjan(int now,int fa) {
    total++;
    int lstnum=0;
    low[now]=dfn[now] = ++cnt;
    s.push(now);
    for (int i = 0; i < a[now].size(); i++) {
        if(a[now][i]==fa){
            lstnum++;
            if(lstnum<2){
                continue;
            }
        }//如果有重边：即 A 到 B 有两条边，那么 A 和 B 为同一个连通块
        int child = a[now][i];
        if(dfn[child]){
            low[now]=min(low[now],dfn[child]);
        }
        else {
            tarjan(child,now);
            if (dfn[now]<=low[child]) {
                cnt2++;
                ans[cnt2].push_back(now);  //加入树根/割点！！！ 因为一个割点属于多个分量
                xintu[n+cnt2].push_back(now); //建立圆方树
                xintu[now].push_back(n+cnt2);
                while (s.top() != child) {
                    ans[cnt2].push_back(s.top());
                    xintu[n+cnt2].push_back(s.top());
```

```
                    xintu[s.top()].push_back(n+cnt2);
                    s.pop();
                }
                ans[cnt2].push_back(s.top());  //ans 存储点双联通分量
                xintu[n+cnt2].push_back(s.top());
                xintu[s.top()].push_back(n+cnt2);
                s.pop();
            }
            low[now] = min(low[now], low[child]);
        }

    }
}
```

调用：

```
    for(int i=1;i<=n;i++) {
        if(!dfn[i]) {
            total=0;  //清空总点数，实现下方 dfs
            while(!s.empty()) s.pop();
            if (a[i].empty()) {  //特判孤立点
                cnt2++;
                ans[cnt2].push_back(i);
                continue;
            }

            tarjan(i, -1);
            dfs(i,-1);  //下示 统计删除某点产生的(a,b)对：
        }
    }


void dfs(int now,int fa){
    int ans=0;
    if(now<=n){
        siz[now]=1;
    }else{
        siz[now]=0;  //方点为 0
    }
    for(int i=0;i<xintu[now].size();i++){
        if(xintu[now][i]==fa) continue;
        dfs(xintu[now][i],now);
        ans+=2*siz[now]*siz[xintu[now][i]];
        siz[now]+=siz[xintu[now][i]];
```

```
            }//计算两两子树的圆点 siz 积

    ans+=2*siz[now]*(total-siz[now]);
    anss[now]=ans;
}
```

10. 强连通分量，缩点，KOSARAJU

第一次 DFS，选取任意顶点作为起点，遍历所有未访问过的顶点，并在回溯之前给顶点编号，也就是**后序遍历。**

第二次 DFS，对于反向后的图，以标号最大的顶点作为起点开始 DFS。这样遍历到的顶点集合就是一个强连通分量。对于所有未访问过的结点，选取标号最大的，重复上述过程。

```
void dfs1(int d) {

    flag[d] = 1;
    for (int i = 0; i < a[d].size(); i++) {
        if (flag[a[d][i]] == 0) {
            dfs1(a[d][i]);
        }
    }
    s.push_back(d);
}
void dfs2(int d) {
    tflag[d] = number;
    sum[number]+=1;
    for (int i = 0; i < ta[d].size(); i++) {
        if (tflag[ta[d][i]] == 0) {
            dfs2(ta[d][i]);
        }
    }
}
void suodian(){
    for (int i = 1; i <= n; i++) {
        if (flag[i] == 0) dfs1(i);
    }
    for (int i = n-1; i>=0; i--) {
        if (tflag[s[i]] == 0) {
            number++;
```

```
            dfs2(s[i]);
        }
        }
    }

vector<int> xintu[100005];
void jiantu(){
    for(int i=1;i<=n;i++){
        for(int j=0;j<a[i].size();j++){
            if(tflag[i]!=tflag[a[i][j]]){
                xintu[tflag[i]].push_back(tflag[a[i][j]]);
            }
        }
    }
}调用:
suodian();
jiantu();
```

10-1.TARJAN 做法

```
void tarjan(int now) {  //有向图不需要 father 参数！！！
    s.push(now);
    visi[now] = true;
    low[now]=dfn[now] = cnt++;
    for (int i = 0; i < a[now].size(); i++) {

        int child = a[now][i];
           if (visi[child]) {  //与无向图不同的地方！！！！！
            low[now] = min(low[now], dfn[child]);
        }
        else if (!dfn[child]) {
            tarjan(child);
            low[now] = min(low[now], low[child]);
        }
    }

    if (dfn[now] == low[now]) {  //放在外面！！！！
        while (s.top() != now) {
            id[s.top()] = cnt2;
            visi[s.top()] = false;
            s.pop();
```

```
            num[cnt2]++;
        }
        id[s.top()] = cnt2;
        visi[s.top()] = false;
        s.pop();
        num[cnt2]++;
        cnt2++;
    }
}
```

调用：

```
for(int i=1;i<=n;i++) {
    if(!dfn[i])
         tarjan(i);
}
```

11. DFS、BFS 搜索

11-1.折半 DFS 搜索

```
void dfs(int now,int sum){
    if(sum>S) return;
    if(now>n/2){
       记录；
        return;
    }

    dfs(now+1,sum+a[now]);
    dfs(now+1,sum);
}
void dfs2(int now,int sum){
    if(sum>S) return;
    if(now>n){
        记录或处理；
        return;
    }
    dfs2(now+1,sum+a[now]);
    dfs2(now+1,sum);

}
```

11-2.折半 BFS 搜索

```
void bfs(){
    queue<node> q;
    q.push(start);
    m[start.ss]=0;
    while(!q.empty()){
        node iq=q.front();
        q.pop();
        if(iq.ss==ans){
            return;
        }
        if(m[iq.ss]>7) break;
        for(int i=1;i<=maxn;i++){
               node tmp=iq;
                     //修改 tmp.ss，得到下一个状态
                if(m.find(tmp.ss)!=m.end()) continue;
                m[tmp.ss]=m[iq.ss]+1; //路径长度+1
                q.push(tmp);
            }
        }
    }
}

void bfs2(){
    queue<node> q;
    q.push(end);
    m2[end.ss]=0;
    while(!q.empty()){
        node iq=q.front();
        q.pop();
        if(m.find(iq.ss)!=m.end()){
           合并处理;
            return;
        }
        if(m2[iq.ss]>8) break;
         for(int i=1;i<=maxn;i++){
               node tmp=iq;
                //与上面不同，这里要逆向处理！！
                if(m2.find(tmp.ss)!=m2.end()) continue;
                m2[tmp.ss]=m2[iq.ss]+1;
                q.push(tmp);
```

```
            }
        }
    }
}
```

11-3.双向 BFS 搜索

```
while(!q.empty()){
    int state=q.front();
    q.pop();
    for(int i=1;i<=maxn;i++){
        if(direction[state]==1){
           获得下一个正状态 nexstate
        }else{
           获得下一个逆状态 nexstate（规则和正状态规则相反）
        }
        if(visi[nexstate]){
            if(direction[state]==direction[nexstate]){
                continue;//访问过了
            }
            else{  //成功相遇
               合并答案
                return;
            }
        }
        visi[nexstate]=1;
        direction[nexstate]=direction[state];
        q.push(nexstate);
    }

}
```

五、数学

1. 快速幂

```
long long FastPow(long long a,long long n,const long long p) //求 a 的 n 次方模 p 的值
{
    long long ans=1;
    while(n)
    {
        if(n&1)
            ans=(ans*a)%p;
        a=(a*a)%p;
      n>>=1;
   }
   return ans;
}
```

1-1 数组版

```
while(m){
       if(m&1){
           for(int i=1;i<=n;i++){
               ans[i]=f[ans[i]];
           }
       }
           for(int i=1;i<=n;i++){
               tmp[i]=f[i];    //复制一份
           }
           for(int i=1;i<=n;i++){
              f[i]=tmp[tmp[i]];//倍增一次
           }

       m>>=1;
   }
```

2.矩阵加速

```
//以下求 Concatenate(n) MOD m
struct juzheng{
    void clear(){
        for(int i=0;i<SIZE;i++) {
            for(int j=0;j<SIZE;j++) {
                a[i][j]=0;
            }
        }
    }
    void init(){
        for(int i=0;i<SIZE;i++) {
            a[i][i]=1;
        }
    }
    int a[SIZE][SIZE];
    juzheng operator * (const juzheng& a2)const{
        juzheng tmp;
        tmp.clear();
        for(int i=0;i<SIZE;i++){
            for(int j=0;j<SIZE;j++){
                for(int k=0;k<SIZE;k++){
                    tmp.a[i][j]=(tmp.a[i][j]+(a[i][k]*a2.a[k][j])%m)%m;
                }
            }
        }
        return tmp;
    }
};

juzheng _pow(juzheng tmp,int b){
    juzheng ans;
    ans.clear();
    ans.init();
    while(b){
        if(b&1){
            ans=ans*tmp;  //同样的矩阵，无所谓左右乘
        }
        tmp=tmp*tmp;

        b>>=1;
    }
```

```
    return ans;
}
// Concatenate(n) 是将 1~n 所有正整数 顺序连接起来得到的数。下计算 C(n) mod m
tmp.a[0][1]=tmp.a[1][1]=tmp.a[1][2]=tmp.a[2][2]=1;
for(int i=10;;i*=10){//枚举位数：10 即一位数，100 即两位数
    tmp.a[0][0]=i%m;
    if(n>=i){
        ans=(_pow(tmp,i-i/10))*ans;
    }else{
        ans=(_pow(tmp,n-i/10+1))*ans;//注意是右乘，最后为转移矩阵左乘初始矩阵
        break;
    }
}

cout<<(ans.a[0][1]%m+ans.a[0][2]%m)%m;
```

3. 线性递推求逆元

```
long long inv[maxn];
void getinverse(int n,long long p) //求 1 到 n%p 的逆元存入 inv
{
    inv[1]=1;
    for(int i=2;i<=n;i++)
        inv[i]=p-inv[p%i]*(p/i)%p;
}
```

4. 扩展欧几里得

Ax+by=c **有解条件**：gcd（a，b）整除 c

不定方程的一个特解：x0=x*c/gcd（a，b）

对应齐次线性方程组的通解：**先求最小非零解** ax1+by1=0，**得** x1= b/gcd（a，b）,y1= -a/gcd（a，b）

通解格式：x0+k*x1，y0+k*y1

```
void exgcd(long long a,long long  b,long long &x,long long &y) //求解 ax+by=gcd(a,b)
 {
     if(b==0)
```

```
    {
        x=1;
        y=0;
        return;
    }
    exgcd(b,a%b,y,x);
    y-=a/b*x;
}
```

5.排列组合数、斯特林数等

```
int C(int n,int k){ //不取模版
    if(k==n||k==0) return 1;
    if(k>=n/2) k=n-k;
    int ans=1;
    for(int i=n,j=1;j<=k;i--,j++){
        ans*=i/j;
    }
    return ans;
}

int jie[10000005];  //阶乘

void init(int n,int mod){
    int ans=1;
    for(int i=1;i<=n;i++){
        ans=ans*i%mod;
        jie[i]=ans;
    }
}
int C(int n,int k,int mod){  //取模版
    if(n==k) return 1;
    long long ans=jie[n];
    ans=(ans*pow_(jie[k],mod-2,mod))%mod;
    ans=(ans*pow_(jie[n-k],mod-2,mod))%mod; //逆元
    return ans;
}

void buid_C(int n){  //递推求组合数，可取模
    c[0][0]=1;
    c[1][0]=c[1][1]=1;
    for(int i=2;i<=n;i++){
```

```
        c[i][0]=1;
        for(int j=1;j<=n;j++){
            c[i][j]=(c[i-1][j-1]+c[i-1][j]);
        }
    }
}

int A(int n,int k){
    int ans=1;
    for(int i=n;i>=n-k+1;i--){
        ans*=i;
    }
    return ans;
}

int pow_(int a,int b,int mod){ //取模版
    int ans=1;
    int tmp=a;
    while(b){
        if(b&1){
            ans=(ans*tmp)%mod;
        }
        b>>=1;
        tmp=(tmp*tmp)%mod;
    }
    return ans;
}



int stirling(int k,int n){ //k 球入 n 盒，使得没有盒为空的方案数
    if(n<=0||k<n) return 0;
    if(k==n) return 1;
    return f(k-1,n-1)+f(k-1,n)*n;
}


int f[1000005];
int xinfeng(int n,int p){ //n 封信送 n 个人，不在其位上的数为 n，有几种方式
    if(n==0) return 1;
    if(n==1) return 0;
    if(f[n]) return f[n];
    return f[n]=((n-1)*(xinfeng(n-1,p)%p+xinfeng(n-2,p)))%p;
```

```
}

void katelan(){ //用栈模拟，压入和弹出序列，过程合法的方案数；-1 +1 交替，前序和≥0 的方案数
    C[0]=C[1]=1;//前提条件！！
    for(int i=2;i<=n;i++){
        for(int j=0;j<i;j++){
            C[i]+=C[j]*C[i-j-1];
        }
    }//Cn=C(n,2n)【组合数】/n+1=C(n,2n)-C(n-1,2n)
}
```

5-1.卢卡斯定理

```
//（计算组合数 C，P 为较小质数）
long long p;
 long long jie[100006];
 long long reverse(long long a) //逆元
 {
     long long ans=1;
     long long n=p-2;
     while(n)
     {
         if(n&1)
             ans=(ans*a)%p;
         a=(a*a)%p;
         n>>=1;
     }
     return ans;
 }
 long long f(long long n,long long k)
 {
     if(k>n)
         return 0;
     return (jie[n]* reverse(jie[k])%p*reverse(jie[n-k]))%p;
 }
 long long cnk(long long n,long long k)  //模 p 意义下的组合数
 {
     if(!k)
         return 1;
     return f(n%p,k%p)*cnk(n/p,k/p)%p;
 }
```

6. 欧拉筛

```
bool vis[maxn];
int prime[maxn];
int top=0;
void GetPrime(int n) //筛出小于等于 n 的所有素数，按序存入 prime 数组
{
    for(int i=2;i<=n;i++)
    {
        if(!vis[i])
            prime[top++]=i;
        for(int j=0;j<top&&i*prime[j]<=n;j++)
        {
            vis[i*prime[j]]=true;
            if(i%prime[j]==0)
                break;
        }
    }
}
```

6-1.埃氏筛

```
int flag[100005];
int cnt=0;
int a[100005];

void getprime(int n){
    for(int i=2;i<=n;i++){
        if(flag[i]==0){
            flag[i]=1;
            a[++cnt]=i;
            for(int j=i+i;j<=n;j+=i){
                flag[j]=1;
            }
        }
    }
    }
```

7. 中国剩余定理

```
void exgcd(long long a,long long b,long long &x,long long &y){
    if(b==0){
        x=1;
        y=0;
    }
    else{
        exgcd(b,a%b,y,x);
        y=y-(a/b)*x;
    }
}
long long gcd(long long a,long long b){
    if(b==0) return a;
    else return gcd(b,a%b);
}
int n;
long long m[11];
long long  a[11]; // a 同余 x mod m
long long  t[11];
long long  Mi[11];
long long  M=1;  //M 是所有 m 之积
long long IntChina(){
    long long ans=0;
    long long y=0;
    for(int i=1;i<=n;i++){
        Mi[i]=M/m[i];
        exgcd(Mi[i],m[i],t[i],y);
        t[i]=(m[i]+t[i]%m[i])%m[i];
        for(int j=1;j<=a[i];j++){
            ans=(ans+Mi[i]*t[i])%M;//防止溢出
        }
    }
    return ans;
}
```

8. 容斥原理

8-1.典例：硬币购物

```
//对于每次购买，他带了 di 枚价值为 ci 的硬币，想购买 s 的价值的东西，求方案数。
dp[0]=1;
for(int i=1;i<=4;i++){
    for(int j=c[i];j<100005;j++){
        dp[j]+=dp[j-c[i]];
    }
}


    cin>>d[1]>>d[2]>>d[3]>>d[4]>>s;
    int ans=dp[s];
    for(int i=1;i<=15;i++){
        int cn=0,sum=0;
        for(int j=1;j<=4;j++){
            if(i&(1<<(j-1))) cn++,sum+=c[j]*(d[j]+1);
        }
        if(s<sum) continue;
        if(cn%2==0){
            ans+=dp[s-sum];
        }else{
            ans-=dp[s-sum];
        }
    }
    cout<<ans<<endl;
```

8-2.典例：倍数关系

```
//求数组 c 中数（不存在倍数关系）的倍数中，在[a,b]内的个数
void dfs(int now,__int128 sum, int num){
    if(sum>b) return;
    if(now>cnt){
        if(num==0) return;//没有选
        if(num%2==1){
            ans+=b/sum-(a-1)/sum;
        }
        else ans-=b/sum-(a-1)/sum;   //求 sum 的倍数个数
        return;
    }
```

```
    dfs(now+1,lc(sum,c[now]),num+1);//利用最小公倍数的性质
    dfs(now+1,sum,num);
}
```

8-3.典例：分特产，ULB 问题

```
//M 种特产分给 N 个人，求方案数。
//我们设 g[i]表示刚好有 i 个同学没有土特产，由于每个同学都至少要获得一个土特产，那么显然 g[0]即为我们的答案。
//f[0]=g[0]+g[1]+g[2]+g[3]+g[4]
//f[1]=g[1]+g[2]*2+g[3]*3+g[4]*4
//f[2]=g[2]+g[3]*3+g[4]*4
//f[3]=g[3]+g[4]*4
//f[4]=g[4]
    int flag=-1;
    for(int i=0;i<=n;i++){
         flag=-flag;
         int tmp=1;
         for(int j=1;j<=m;j++){
                tmp=(tmp*c[a[j]+n-i-1][n-i-1])%MOD;  //a[j]个物品分给 n-i 个人，有人可以不分到物品，有多少种方案：ULB 问题
         }
         tmp=(tmp*c[n][i])%MOD;  //选这 i 个人
         ans=(ans+flag*(tmp)+MOD)%MOD;//容斥定理！
    }
```

9.扩展欧拉定理

```
void init(int n){    //数组版
    phi[1]=1;
    for(int i=2;i<=n;i++){
        if(!phi[i]){
            prime[++cnt]=i,phi[i]=i-1;  //prime 存的是质数
        }
        for(int j=1;j<=cnt&&i*prime[j]<=n;j++){
            if(i%prime[j]==0){
                phi[i*prime[j]]=phi[i]*prime[j];
                break;
            }
            else{
                phi[i*prime[j]]=phi[i]*(prime[j]-1);
```

```
            }
        }
    }
}


void init(int n){

    phi[1]=1;

    for(int i=2;i<=n;i++){

        if(!phi[i]){

            prime[++cnt]=i,phi[i]=i-1;  //prime 存的是质数

        }

        for(int j=1;j<=cnt&&i*prime[j]<=n;j++){

            if(i%prime[j]==0){

                phi[i*prime[j]]=phi[i]*prime[j];

                break;

            }

            else{

                phi[i*prime[j]]=phi[i]*(prime[j]-1);

            }

        }

    }

}

long long phi(long long n){
    long long tmp=n;
    for(int i=2;i*i<=n;i++){
        if(n%i==0){
            tmp=tmp/i*(i-1);
            while(n%i==0) n/=i;
```

```
        }
    }
    if(n!=1) tmp=tmp/n*(n-1);
    return tmp;
}

long long q(long long n,long long e,long long p){  求 a 的 b 次方 mod m
    n%=p;
    long long ans=1;
    long long pow=n;
    while(e){
        if(e&1){
            ans*=pow;
            ans%=p;
        }
        e>>=1;
        pow=(pow*pow)%p;
    }
    return ans;
}
```

调用：

```
p=phi(m);
b=input();
if(b≥p) b=b%p+p;  //注意，当 a 和 m 互素时，b 取 b%p。
cout<<q(a,b,m);
```

9-1.欧拉定理应用

统计 x,y≤n 中有多少对数的 gcd 为质数：

```
init(n);
int ans=0;
for(int i=1;i<=n;i++){//前缀和
    sum[i]=sum[i-1]+phi[i];
}
for(int i=1;i<=cnt;i++){  //令 x=x’*p,1≤x’≤n/p，枚举这个 p
    ans=(ans+2*(sum[n/prime[i]])-1); //计算 x，y 对。-1 是因为(x=gcd,y=gcd)被计算了两次
}
```

计算 gcd（x，n）的和，x<=n：

```
for(int i=1;i*i<=n;i++){
        if(n%i==0){
            ans+=i*phi(n/i);
            if(i*i!=n){
                ans+=(n/i)*phi(i);  原理：gcd(x/i，n/i)=1
            }
        }
    }
```

10.整除分块

```
//快速找 flour（a/i）相同的区间

 for(int l=1,r=1;l<=n&&r<=n;  l=r+1;){//计算  i*flour（k/i），1≤i≤n 的和

        int tmp=k/l;

        if(tmp==0) r=n;

        else r=min(k/tmp,n);

        ans+=tmp*(r-l+1)*(l+r)/2;  //俩值相乘:值乘以 l 到 r 范围的求和

}
```

求 x/y 为有限小数的对数。//可分解为 ac/bc，c 不包含 2、5 因子，b 只包含 2、5 因子。最终可化简为 e*f（n/c）*flour(n/c)，即满足条件的 c 的个数，乘以满足条件的 b 的个数，再乘以 a 的个数

```
void init(){
    for(int i=1;i<=n;i*=2){
        for(int j=i;j<=n;j*=5){
          x[++cnt]=j;
        }
    }
    cnt= unique(x+1,x+1+cnt)-x-1;
    sort(x+1,x+1+cnt);
}
for(int l=1,r=1;l<=n&&r<=n;l=r+1){  //枚举 c
        int t=n/l;
        if(t==0) r=n;
        else r=n/t;//除法分段
        while(coun>=1&&x[coun]>t) coun--;//满足单调性：随着 c 的增加，满足条件的 a 会变小
        int tmp=r-l+1;
        tmp-=r/2-(l-1)/2;
        tmp-=r/5-(l-1)/5;
        tmp+=r/10-(l-1)/10;//容斥，求出 c 的个数（不含 2、5 因子）
        ans+=coun*tmp*t;//在这个区间内，b 的个数，乘以 c 的个数，乘以 a 的个数
    }
```

11. 均值，PSU 问题

```
// 连续的 X 个 1 可以贡献 X³ 的分数，每格有概率 p 成功，求总分数期望

 for(int i=1;i<=n;i++){

        a[i]=(a[i-1]+1)*p[i];  //第 i 位为 1 的期望，不考虑为 0          x

        b[i]=(b[i-1]+2*a[i-1]+1)*p[i]; //第 i 位为 1 的期望，不考虑为 0    x²

        c[i]=(c[i-1]+3*b[i-1]+3*a[i-1]+1)*p[i]+c[i-1]*(1-p[i]); //第 i 位的期望  x³

    }
```

12. 高斯消元

```
 int line=1;
void gaosixiaoyuan(){
    for(int j=1;j<=n;j++){
        for(int i=line;i<=n;i++){
            if(fabs(gaosi[i][j])>eps) {
                for(int s=1;s<=n+1;s++){
                    swap(gaosi[line][s],gaosi[i][s]);
                }
                break;
            }
            if(i==n) goto cn;//这一列没有非零元素，无法消元
        }

        for(int i=n+1;i>=j;i--){//注意是倒序
            gaosi[line][i]/=gaosi[line][j];
        }
        for(int i=1;i<=n;i++){
            if(i!=line){
                for(int k=n+1;k>=j;k--){//注意是倒序
                    gaosi[i][k]-=gaosi[line][k]*gaosi[i][j]; //消元
                }
            }
        }
        line++;
        cn:;
```

```
    }

}

if(line<=n){
    for(int i=line;i<=n;i++){
        if(fabs(gaosi[i][n+1])>eps){
            cout<<"-1";   //无解；一边为0，一边不为0
            return 0;
        }
    }
    cout<<"0";  //无穷多解
    return 0;
}
for(int i=1;i<=n;i++){
    cout<<"x"<<i<<"=";
    printf("%.2f\n",gaosi[i][n+1]);  //有限解
}
```

12-1.行列式求值

```
int flag=1;

void hanglieshi(){

    for(int i=1;i<=n;i++){

        for(int j=i+1;j<=n;j++){

            while(gaosi[i][i]){

                int x=gaosi[j][i]/gaosi[i][i];

                for(int k=i;k<=n;k++){

                    gaosi[j][k]=(gaosi[j][k]-x*gaosi[i][k]%m+m)%m;

                }

                swap(gaosi[i],gaosi[j]);

                flag=-flag;
```

```
            }

            swap(gaosi[i],gaosi[j]);

            flag=-flag;//辗转消元法：保证精度问题

        }

    }

}
```

调用：

```
  hanglieshi();

    for(int i=1;i<=n;i++) sum=(sum*gaosi[i][i])%m;

    cout<<(sum*flag%m+m)%m;
```

13.异或线性基

```
//以下求线段树的每个节点都是一个线性基，并对多个线性基进行合并的情况
#include <bits/stdc++.h>
using namespace std;
const int maxn=1e5+5;
const int maxv=30;
vector<int> G[maxn];
int siz[maxn],dfn[maxn],cnt=1;
void dfs(int now,int father)
{
    siz[now]=1;
    dfn[now]=cnt++;
   for(int i=0;i<G[now].size();i++){
       int to=G[now][i];
       if(to==father) continue;
       dfs(to,now);
       siz[now]+=siz[to];
   }
}

class Hamel   //线性基
{
public:
    int p[maxv]={0}; //maxv 为位数上限
```

```
int d_[maxv]={0};
bool have_zero=false;
bool operator==(const Hamel &that) const
{
    for (int i=0;i<maxv;i++)
    {
        if (p[i]==0&&that.p[i]!=0)
            return false;
        if (p[i]!=0&&that.p[i]==0)
            return false;
    }
    return true;
}

void rebuild(){
    cnt=0;
    for(int i=maxv-1;i>=0;i--){
        for(int j=i-1;j>=0;j--){
            if(p[i]&(1ll<<j)){
                p[i]=p[i]^p[j];   //求最简型；求 getmin rank findkth 使用
            }
        }
    }
    for(int i=0;i<maxv;i++){
        if(p[i]) d_[cnt++]=p[i];
        p[i]=0;
    }
    for(int i=0;i<cnt;i++){
       p[i]=d_[i];
    }
}


bool insert(int x) //贪心方法；将 x 插入线性基
{
    bitset<maxv> b(x);
    for (int i = maxv-1; i >= 0; i--)
    {
        if(b[i]){
            if(p[i]){
                b^=p[i];
            }else{
                p[i]=b.to_ulong();
                return true;
```

```
            }
        }
    }
    have_zero=true;
    return false;
}

int findkth(int k) //查找第 k 小
{
    if(have_zero) k--; //线性基无法计算出 0
    if(k==0) return 0;  //特殊处理 0
    if(k>=(1ll<<cnt)){
        return -1; //超出最大范围
    }
    int ans=0;
    bitset<maxv> b(k);
    for(int i=60;i>=0;i--){
        if(b[i]){
            ans^=p[i];
        }
    }

    return ans;

}

int ask(long long x){  //一个数能否异或出来
    for(int i=maxv-1;i>=0;i--){
        if(x&(1ll<<i)){
            x=x^p[i];
        }
    }
    return x==0;
}
long long get_max(){  //最大值是第(1ll<<cnt)-!have_zero 小，当需要处理 0 时
    long long ans=0;
    for(int i=maxv-1;i>=0;i--){
        if((ans^p[i])>ans){
            ans=ans^p[i];
        }
    }
    return ans;
}
long long get_min(){
```

```
        if(have_zero){
            return 0;
        }
        for(int i=0;i<maxv;i++){
            if(p[i]!=0){
                return p[i];
            }
        }
        return -1;
    }

    long long rank(long long x){
        long long ans=0;
        for(int i=60;i>=0;i--){
            if(x>=p[i]){
                ans+=(1ll<<i);
                x^=p[i];
            }
        }
        return ans+have_zero;
    }

    void clear() //清空
    {
        for (int i = 0; i < maxv; i++)
            p[i] = 0,d_[i]=0;
        cnt=0;
        have_zero= false;
    }
    Hamel operator*(const Hamel &that)
    {
        Hamel ans=*this;
        for (int i=maxv-1;i>=0;i--)
            if (that.p[i])
                ans.insert(that.p[i]);
        return ans;
    }
};
class SegmentTree
{
public:
    Hamel a[4*maxn];
    bool push_up(int rt) //向上更新
    {
```

```
        auto nxt=a[rt*2]*a[rt*2+1];
        if (nxt==a[rt])
            return false;
        a[rt]=nxt;
        return true;
    }
    bool updata1(int loc, int k, int l, int r, int rt) //线段树区间为从 l 到 r，把区间 x 到 y 每个数
+k
    {
        if (l==r)
            return a[rt].insert(k);
        int mid = (l + r) / 2;
        bool ans=false;
        if (loc <= mid)
            ans=updata1(loc, k, l, mid, rt * 2);
        else
            ans=updata1(loc, k, mid + 1, r, rt * 2 + 1);
        if (ans)
            return push_up(rt);
        return ans;
    }
    void build(int l, int r, int rt) //从 l 到 r 建立线段树
    {
        a[rt].clear();
        if (l == r)
        {


            return;
        }
        int mid = (l + r) / 2;
        build(l, mid, rt * 2);
        build(mid + 1, r, rt * 2 + 1);
    }
    Hamel query(int x, int y, int l, int r, int rt) //线段树区间为从 l 到 r，询问区间 x 到 y 的和
    {
        if (x <= l && y >= r)
            return a[rt];
        int mid = (l + r) / 2;
        Hamel ans;
        if (x <= mid)
            ans = query(x, y, l, mid, rt * 2);
        if (y > mid)
            ans =ans* query(x, y, mid + 1, r, rt * 2 + 1);
        return ans;
```

```
    }
}st;
signed main()
{
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);
    int t;
    cin>>t;
    while(t--)
    {
        int n,m,u,v;
        cin>>n>>m;
        for (int i=1;i<=n;i++)
            G[i].clear();
        st.build(1,n,1);
        cnt=1;
        for (int i=1;i<n;i++)
        {
            cin>>u>>v;
            G[u].push_back(v);
            G[v].push_back(u);
        }
        dfs(1,0);
        int op,x,y;
        while (m--)
        {
            cin>>op>>x>>y;
            if (op==1)
                st.updata1(dfn[x],y,1,n,1);
            else
            {
                auto ans=st.query(dfn[x],dfn[x]+siz[x]-1,1,n,1);
                ans.rebuild();
                cout<<ans.findkth(y)<<'\n';
            }
        }
    }
}
```

14.质因数分解

```
void get_prim(int tmp){
```

```
    prim.clear();
    for(int i=2;i*i<=tmp;i++){
        if(tmp%i==0){
            while(tmp%i==0){
                tmp=tmp/i;
            }
            prim.push_back(i);
        }
    }
    if(tmp!=1){
        prim.push_back(tmp);
    }
}

bool is_prim(int a){
    if(a==2) return true;
    for(int i=2;i*i<=a;i++){
        if(a%i==0){
            return false;
        }
    }
    return true;
}
```

14-1.约数计数

```
求a方的约数个数

void jishu(){
    for(int i=2;i*i<=a;i++){
        if(a%i==0){
            tmp=0;
            while(a%i==0) tmp++1,a/=i;
            ans*=tmp*2+1;
        }
    }
    if(a!=1) ans*=3;
}
```

15. 三分

```
double find(double l,double r) //求 l 到 r 单峰函数的极大值点
{
    double ans;
    while(r-l>eps)
    {
        double mid=(l+r)/2;
        double lmid=mid-eps,rmid=mid+eps;
        if(f(lmid)>f(rmid))
        {
             ans=mid;
             r=mid;
         }
         else
             l=mid;
     }
     return ans;
 }
```

六、 dp 示范

最优子结构：问题的最优解所包含的子问题的解也是最优的。
无后效性：当前阶段的求解只与之前阶段有关，而与之后的阶段无关。
决策并不是线性的，而需要全面考虑不同情况，分别决策。
步骤：寻找子问题、定义状态、导出状态转移方程、确定边界条件。

0. 纸币问题

```
void dp(){
    //至少多少纸币可以凑够 w 元：
    for(int i=1;i<=w;i++){
        for(int j=1;j<=n;j++) {
            if (i >= v[j]) {
                dp[i] = min(dp[i], dp[i - v[j]] + 1);
            }
        }
    }

    //有多少种纸币组合可以凑够 w 元：
    for(int i=1;i<=n;i++) {
```

```
    for(int j=v[i];j<=w;j++){
            dp[j] = dp[j]+dp[j - v[i]];
        }
    }
}
```

1.字符串 dp

```
//求能由一些小串组成的大串的最大长度
   dp[0]=1;
    int ans=0;
    for(int i=1;i<s.size();i++){
        for(int j=i;j>=1;j--){
            string tmps=bs.substr(j,i-j+1);
            if(dp[j-1]&&s.find(tmps)!=s.end()){ //字符集合存放在 s 中
                dp[i]=1;
                ans=max(ans,i);
                break;
            }
        }
    }

   //两个字符串，可以增，删，改，最少要操作几个字符才能相同?
void dp() {
          for(int i=1;i<=lena;i++)
            f[i][0]=i;
           for(int i=1;i<=lenb;i++)
            f[0][i]=i;
          for(int i=1;i<=lena;i++){
              for(int j=1;j<=lenb;j++) {
                if(a[i-1]==b[j-1]) {
                    f[i][j]=f[i-1][j-1];
                    continue;
                }
                    f[i][j]=min(min(f[i-1][j],f[i][j-1]),f[i-1][j-1])+1;//分别对应增，删，改
              }
          }
 }

//从字符串 A 中，选取 k 个块拼成 B，有几种方式
dp[0][0][0][0]=1;
```

```
 dp[1][0][0][0]=1;
for(int i=1;i<=n;i++){
    for(int j=1;j<=m;j++){
        for(int d=1;d<=k;d++){// 第四维表示 A 串的 i 个字符是否被取。
            if(s1[i]==s2[j]){
                dp[i][j][d][1]=(dp[i-1][j-1][d][1]+dp[i-1][j-1][d-1][0]+dp[i-1][j-1][d-1][1])%MOD;
                dp[i][j][d][0]=(dp[(i-1)][j][d][1]+dp[(i-1)][j][d][0])%MOD;
            }
            else{
                dp[i][j][d][1]=0;
                dp[i][j][d][0]=(dp[(i-1)][j][d][1]+dp[(i-1)][j][d][0])%MOD;
            }
        }
    }
}

 cout<<(dp[n][m][k][1]+dp[n][m][k][0])%MOD;
```

2. 背包 dp

2-1.典例：聪明的奶牛

```
for(int i=0;i<=800000;i++){ //智力和大于 0，情商也要大于 0，两者和最大多少
    dp[i]=-1e9;
}
dp[400000]=0;//偏移量
for(int i=1;i<=n;i++){
    if(a[i].zhi>=0){
        for(int j=800000;j>=a[i].zhi;j--){
            dp[j]=max(dp[j],dp[j-a[i].zhi]+a[i].qing);
        }
    }
    else{
        for(int j=0;j<=800000+a[i].zhi;j++){
            dp[j]=max(dp[j],dp[j-a[i].zhi]+a[i].qing);
        }
    }
}
int ans=-1e9;
for(int j=400000;j<=800000;j++){ //枚举大于 0 的智力
```

```
    if(dp[j]>=0){//里面存着情商
        ans=max(ans,dp[j]+j-400000);
    }
}
```

2-2.分组背包 DP

```
 for(int i=1;i<=n;i++)
 for(int j=t;j>=tt[i];j--){
                for(int k=0;k<=num[i];k++) {
                    if(j>=k*tt[i])
                    dp[j] = max(dp[j], dp[j - k*tt[i]] + k*c[i]);
                }
            }
```

3. 区间 dp

3-1.典例：祖玛

```
for(int i=1;i<=n;i++){ //不断移除回文字串，最多要几次
    scanf("%d",&a[i]);
    dp[i][i]=1;
    if(a[i]==a[i-1]) dp[i-1][i]=1;
    else dp[i-1][i]=2;
}

for(int t=2;t<=n;t++){
    for(int i=1;i<=n-t;i++){
        int j=i+t;
        if(a[i]==a[j]) dp[i][j]=min(dp[i][j],dp[i+1][j-1]);

        for(int k=i;k<j;k++){
            dp[i][j]=min(dp[i][j],dp[i][k]+dp[k+1][j]);
        }

    }
}
```

3-2.典例：大爷关灯

```
//有一个大爷在 c 点，左右去关灯，关灯前持续消耗功率
dp[c][c][0]=dp[c][c][1]=0;
for(int step=2;step<=n;step++){
    for(int i=1;i+step-1<=n;i++){
        int j=i+step-1;//0 表示区间向左扩大
        dp[i][j][0]=min(dp[i+1][j][0]+(a[i+1].wei-a[i].wei)*(sum[i]+sum[n]-
sum[j]),dp[i+1][j][1]+(a[j].wei-a[i].wei)*(sum[i]+sum[n]-sum[j])); //sum 是功率的前缀和
        dp[i][j][1]=min(dp[i][j-1][0]+(a[j].wei-a[i].wei)*(sum[i-1]+sum[n]-sum[j-1]),dp[i][j-
1][1]+(a[j].wei-a[j-1].wei)*(sum[i-1]+sum[n]-sum[j-1]));

    }
}
cout<<min(dp[1][n][1],dp[1][n][0]);
```

3-3.典例：手串（矩阵优化)

环状图，任意 m 个连续的地方，不得超过 k 个 1

```
 cin>>n>>m>>k;
    juzheng b;
    for(int i=0;i<(1<<m);i++){//枚举新状态
        if(cnt(i)>k) continue;
        b.a[i>>1][i]=1;  a[10]可由 a[01]、a[11]转化来，在对应位置填 1。行表示上一个状态，列表示下
一个状态
        if(cnt(i>>1)<=k-1){
            b.a[(i>>1)|(1<<(m-1))][i]=1;
        }
    }
    juzheng ans=pow(b,n); //注意：先预定 0 号点的状态，它与 n 号点相同。故总共需要 n 次方。
    int tans=0;
    for(int i=0;i<(1<<m);i++){
        if(cnt(i)>k) continue;
        tans=(tans+ans.a[i][i])%MOD;
    }
    cout<<tans;
}
```

3-4.典例：合数

```
  //相邻两相同数合成大 1 数。以下利用倍增思想
 for(int i=1;i<=n;i++){
      cin>>a[i];
      dp[a[i]][i]=i+1;
  }
  int ans=0;
 for(int i=1;i<100;i++){
     for(int j=1;j<=n;j++){
         if(dp[i-1][dp[i-1][j]]!=0) dp[i][j]=dp[i-1][dp[i-1][j]];
         if(dp[i][j]!=0) ans=max(ans,i);
     }
 }
```

4. 树上 dp

4-1.树上背包 DP

```
//现在这颗树枝条太多了，需要剪枝。但是一些树枝上长有苹果。给定需要保留的树枝数量，求出最多能留住多少苹果。

for(int i=0;i<son[now].size();i++){

    for(int j=m;j>0;j--){  //分给这个节点多少树权

        for(int k=0;k<=j-1;k++){ //当前遍历到的儿子拿走几个树权

            dp[now][j]=max(dp[now][j],dp[now][j-k-1]+v[son[now][i]]+dp[son[now][i]][k]);

        }

    }

}


//总重量为 m，求价值最大值。
void f(int now){
```

```
    for(int i=0;i<xintu[now].size();i++){
        f(xintu[now][i]);
        for(int j=m;j>=0;j--){
            for(int k=0;k<=j-neww[xintu[now][i]];k++){
                dp[now][j]=max(dp[now][j],dp[now][j-k-
neww[xintu[now][i]]]+dp[xintu[now][i]][k]+newv[xintu[now][i]]);
            }
        }

    }
}
//三部分：分给前面节点 dp 的重量，分给当前子节点 dp 的重量，该节点引入当前子节点所获得的孤点价值
```

4-2 树上路径 DP

```
求树上值最大的路径（边权含负值）
void dfs(int now,int fa){
    dp[now][0]=dp[now][1]=0;
    for(int i=0;i<vv[now].size();i++){
        int son=vv[now][i].to;
        int dis=vv[now][i].dis;
        if(son==fa) continue;
        dfs(son,now);
        if(dp[son][0]+dis>dp[now][0]){
            dp[now][1]=dp[now][0];
            dp[now][0]=dp[son][0]+dis;
        }else if(dp[son][0]+dis>dp[now][1]){
            dp[now][1]=dp[son][0]+dis;
        }
    }
    ans=max(ans,max(dp[now][0],dp[now][0]+dp[now][1]));
}
```

4-3.换根 DP

```

void dfs(int now,int father){
    son[now]=1;
    for(int i=0;i<v[now].size();i++){
        if(v[now][i]==father) continue;
        dfs(v[now][i],now);
```

```
        son[now]+=son[v[now][i]];
        ff[now] += ff[v[now][i]] + son[v[now][i]]; //求到根节点的路径之和 ff[1]
    }

}
void f(int now,int father){
    for(int i=0;i<v[now].size();i++){
        if(v[now][i]==father) continue;
        dp[v[now][i]]= dp[now]-son[v[now][i]]+n-son[v[now][i]]; //求到某点的路径之和
        f(v[now][i],now);
    }
}
```

5. 状压 dp

5-1.典例：吃奶酪

```
//房间里放着 nn 块奶酪。一只小老鼠要把它们都吃掉，问至少要跑多少距离？
   for(int i=1;i<=n;i++){
        dp[i][(1<<(i-1))]=dis[0][i];
    }
    for(int k=0;k<=(1<<n)-1;k++){
        for(int i=1;i<=n;i++){
            for(int j=1;j<=n;j++){
                if(i==j) continue;
                if((k&(1<<(i-1)))&&(k&(1<<(j-1)))){    【注意：对 j 的限制可省略！！】
                    dp[i][k]=min(dp[i][k],dp[j][k-(1<<(i-1))]+dis[j][i]);
                }
            }

        }
    }
```

5-2.典例：炸鱼

```
//炸弹可以十字形爆开，炸掉所有格子的鱼，最少要多少个炸弹？
for(int i=originstate;i>=0;i--){  //初始状态压每个格子的鱼数目(0,1,2,3)
          if(dp[i]==inf) //非合法状态
                    continue;
          for(int j=0;j<bomb.size();j++){ //每次多选择一个炸弹
                    //枚举炸鱼位置开炸
```

```
            nextstate=i;
            //对被炸到的有鱼位置都减一
            for(int z=0;z<bomb[j].arr.size();z++){//枚举炸弹能炸到的鱼塘位置
                        if(check(i,bomb[j][z])) //如果这个鱼塘还有鱼
                                    nextstate-=1*pow4[bomb[j].arr[z]];
                        }
                        dp[nextstate]=min(dp[i]+1,dp[nextstate]);
            }
}
```

6. 其他 dp

6-1.典例：道路游戏

```
//同一时间最多一个机器人，最多走 k 条边，硬币随时间分布，求 m 秒最多收益。
   for(int i=1;i<=n;i++) cin>>c[i];//买机器人的费用
   for(int i=1;i<=m;i++){//枚举秒数
       for(int j=1;j<=n;j++){
           int start=dp[i-1]-c[j];
           for(int s=0;s<k&&i+s-1<=m;s++){
               start+=a[(j+s-1)%n+1][i+s];//获得的硬币数
               dp[i+s]=max(dp[i+s],start);//取最大值
           }
       }
   }
  cout<<dp[m];
```

6-2.典例：买股票（单调队列优化）

```
第 i 天的股票买入价为每股 APi，第 i 天的股票卖出价为每股 BPi（数据保证对于每个 i，都有 APi≥BPi），但是每天不能无限制地交易，于是股票交易所规定第 i 天的一次买入至多只能购买 ASi 股，一次卖出至多只能卖出 BSi 股。两次交易最少相隔 w 天。
for(int j=0;j<=a[i].as;j++){
    dp[i][j]=-j*a[i].ap; //初始值
}
for(int j=0;j<=maxp;j++) {
    dp[i][j] = max(dp[i][j],dp[i-1][j]); //不买不卖
}
if(i-w<=1) continue;
```

```
deque<node2> q;
for(int j=0;j<=maxp;j++){   //j 为总股票；买入：f[i,j]=max(f[i,j] ，f[i-w-1,k]+k×api)-j×api
    while(!q.empty()&&j-q.front().gupiao>a[i].as){
        q.pop_front();//当天与 i-w-1 天最多相差 as 张股票
    }
    while(!q.empty()&&dp[i-w-1][j]+j*a[i].ap>q.back().dp+q.back().gupiao*a[i].ap){
        q.pop_back();//即表达式，保证队列元素不递增
    }
    q.push_back({dp[i-w-1][j],j});
    dp[i][j]=max(dp[i][j],q.front().dp+q.front().gupiao*a[i].ap-j*a[i].ap);

}
//卖出时，j 倒序遍历
```

6-3.典例：午饭

```
N 行 m 列，值为 a[i][j]，第 i 行最多选值 1，求存在一列的选数和＞总和一半的方案数。
for(int w=1;w<=m;w++) {//枚举过半的那列
    memset(dp,0,sizeof(dp));
    dp[0][n]=1;//即 dp[0][0]=1;
    for (int i = 1; i <= n; i++) {//前 i 行
        for (int j = -i+n; j <= i+n; j++) {//当前选定列的值 比其他列之和 最多 多 i 或者最少 少 i
            dp[i][j]=(dp[i-1][j]+(dp[i-1][j-1]*a[i][w])%MOD+(dp[i-1][j+1]*(1[i]-
a[i][w]))%MOD)%MOD;不选当前行+选择选定列+选定其他列
            if(i==n&&j>n) ans=(ans+dp[i][j])%MOD;//统计答案

        }

    }
}//暴力：fi,j,k 表示对于 col 这一列，前 i 行在 col 列中选了 j 个，在其他列中选了 k 个，那么令 si 为第 i
行的总和，则有转移：f[i,j,k]=f[i-1,j,k]  +  a[i,col]*f[i-1,j-1,k]  +  (si-a[i,col])*f[i-1,j,k-
1]
```

6-4.典例：游戏

```
//n 个格子有 k 个连续块的方案数，n、k 为 1e5
int main(){
    f[k]=1;
    for(register int i=k+1;i<=n;++i){
        f[i] = 2*f[i-1] + pow(2,i-k-1)- f[i-k-1];  //随意 01+仅这段做出贡献
```

```
    }
    printf("%d",f[n]);
}
```

6-5.典例：拆分

```
//给定一个整数 n，求将 n 分解为互不相同的不小于 2 的整数的乘积的方案数。
for(int i=1;i<=n/i;i++){
    if(n%i==0){
        a[++cnt]=i;
        if(i*i!=n){
            a[++cnt]=n/i;
        }
    }
}//Ologn 统计因数。注意，一个数的因数，其因数也是这个数的因数。
sort(a+1,a+1+cnt);
for(int i=1;2*i<=cnt+1;i++){
    pos1[a[i]]=i;
    pos2[a[i]]=cnt-i+1;  //优化：将取值范围优化为 log（MAXN）
}
memset(dp,0,sizeof(dp));
dp[1][1]=1;//要构成第 1 个数，利用前 1 给数，有 1 种方案
for(int i=1;i<=cnt;i++){
    for(int j=1;j<=cnt;j++){
        dp[i][j]= (dp[i][j]+dp[i][j-1])%MOD; //不使用第 j 个数
        if(a[i]%a[j]==0){
            dp[i][j]=(dp[i][j]+dp[a[i]/a[j]<= sqrt(n)?pos1[(a[i]/a[j])]:pos2[n/(a[i]/a[j])]][j-
1])%MOD; //使用第 j 个数，利用 pos[]找到下标并转移
        }

    }
}
cout<<dp[cnt][cnt]-1<<endl;
```

七、其它

4.反悔贪心

```
    T1 为抢修时间，t2 为截止时间。求最多维修量
for(int i=1;i<=n;i++){//按 t2 排序
        if(now+a[i].t1<=a[i].t2){
```

```
            ans++;
            q.push(a[i]);//q 为优先队列，按 t1 排序
            now+=a[i].t1;
        }
        else{
            if(a[i].t1<q.top().t1) {
                now =now- q.top().t1+a[i].t1;
                q.pop();
                q.push(a[i]);
            }
        }
    }
```

1. 关闭同步流

```
ios::sync_with_stdio(false);
cin.tie(nullptr);
cout.tie(nullptr);
```

2. 快读

```
inline long long input(){
    long long n=0;
    int f=1;
    char c=getchar();
    while(c<'0' || c>'9'){
        if(c=='-') f=-1;
        c=getchar();
    }
    while(c>='0' && c<='9'){
        n=(n<<3)+(n<<1)+(c^48);
        c=getchar();
    }
    return n*f;
}
```

3. 数列离散化

```
int a[maxn],d[maxn];
void discrete(int n) //将 0-n-1 的数离散化放入 d
```

```
 {
     for(int i=0;i<n;i++)
         d[i]=a[i];
     sort(a,a+n);
     n=unique(a, a + n)-a;
     for (int i = 0; i < n; i++)
         d[i] = lower_bound(a, a + nn, d[i]) - a + 1;
 }

//离散化写法 2
  for(int i=1;i<=n;i++){
        cin>>a[i].val;
        a[i].num=i;
    }
    sort(a+1,a+1+n);
    for(int i=1;i<=n;i++){
        ran[a[i].num]=i;  //ran[1]即原数列第 1 个数离散化后的值。
    }
```

3-1.二维离散化

```
//OVERPLANTING 问题
    for(int i=1;i<=n;i++){//坐标系内有多个可重叠矩形，求其面积之和
    cin>>a[i].x1>>a[i].y1>>a[i].x2>>a[i].y2;
    b[++cnt]=a[i].x1;
    b[++cnt]=a[i].x2;
    b[++cnt]=a[i].y1;
    b[++cnt]=a[i].y2;
}

sort(b+1,b+1+cnt);
int cn= unique(b+1,b+1+cnt)-b-1;
for(int i=1;i<=cn;i++){
    m[b[i]]=i;
}
for(int i=1;i<=n;i++){
    for(int j=m[a[i].x1];j<m[a[i].x2];j++){
        f[j][m[a[i].y2]]++;
        f[j][m[a[i].y1]]--;
    }
}
```

```
for(int i=1;i<=cn;i++){
    for(int j=1;j<=cn;j++){
        f[i][j]+=f[i][j-1];
    }
}

long long ans=0;
for(int i=1;i<=cn;i++){
    for(int j=1;j<=cn;j++){
        if(f[i][j]) ans+=(long long)(b[i+1]-b[i])*(b[j+1]-b[j]);
    }
}
cout<<ans;
```

4. 双指针

```
for(int i=1,j=1;i<=n;i++){   //一次需要包含 num 个不同的 id，求最小的距离差
    if(i>=2){
        number[a[i-1].id]--;
        if(number[a[i-1].id]==0){
            number[0]--;
        }
    }
    while(j<=n){
        if(number[0]==num){
            break;
        }
        number[a[j].id]++;
        if(number[a[j].id]==1){
            number[0]++;
        }
        j++;
    }
    if(number[0]==num){
        ans=min(ans,a[j-1].x-a[i].x);
    }
}
//求离自己第 k 近的数
   f[1]=k+1;
    int l=1,r=k+1;
    for(int i=2;i<=n;i++){
        while(r+1<=n&&a[i]-a[l]>a[r+1]-a[i]) l++,r++;
```

```
        if(a[i]-a[l]>=a[r]-a[i]){
            f[i]=l;
        }
        else f[i]=r;
    }
```

5.根号分治

给你一个长度为 5×1e5 的序列，初值为 0 ，你要完成 q 次操作，操作有如下两种：

1 x y: 将下标为 x 的位置的值加上 y。

2 x y: 询问所有下标模 x 的结果为 y 的位置的值之和。

```
#include<bits/stdc++.h>
#define ll long long
#define mp make_pair
using namespace std;
ll sum[755][755],a[500005];  //这里我的阈值取了700：根号5e5
int main(){
          ios_base::sync_with_stdio(false);cin.tie(0),cout.tie(0);
          int q;cin>>q;
          for(;q--;){
                    int tp,x,y;cin>>tp>>x>>y;
                    if(tp==1){
                              for(int i=1;i<700;++i)sum[i][x%i]+=y;   //枚举模数
                              a[x]+=y;
                    }else{
                              if(x<700){
                                        cout<<sum[x][y]<<endl;
                              }else{
                                        ll rt=0;
                                        for(int i=y;i<=500000;i+=x)rt+=a[i];  //暴力统计
                                        cout<<rt<<endl;
                              }
                    }
          }
}
```

6. 单调队列

```
for(int i=1;i<=n;i++) { //求长度为 k 个数的最大值，建立非递增队列
          if (!s.empty() && s.front() + k <= i) s.pop_front();
          scanf("%d", &a[i]);
          while (!s.empty() && a[i] > a[s.back()]) s.pop_back();
          s.push_back(i);
          if (i >= k) {
                    printf("%d ", a[s.front()]);
          }
}
```

7. 单调栈 H

```
for(int i=1;i<=n+1;i++){  //求每个数后面第一个大于自己的数；建立非递增栈
        while(!s.empty()&&d[s.top()]<d[i]){
            m[i].push_back(s.top());
            s.pop();

        }
        s.push(i);
```

8. 迭代器等

```
//map 迭代器
map<int,int> m;
map<int,int>::iterator it;
m.insert({1,2});
it=m.lower_bound(2); //第一个≥2 的元素的迭代器位置
it=m.upper_bound(2);//第一个＞2 的元素的迭代器位置

//map 删除
for(it=m.begin();it!=m.end();){
    if(it->second==val){
        m.erase(it);
    }
    else{
        it++;
    }
}
```

```
//vector 删除
vector<int> v;
v.insert(v.begin()+2,-1);
v.erase(v.begin(),v.end());

//固定数组、vector 查询
int a[5]={0,1,2,3,4};
cout<<upper_bound(a+1,a+5,2)-a; //返回下标，从 1 开始
vector<int> v;
v.push_back(0),v.push_back(1),v.push_back(2),v.push_back(3);
cout<<upper_bound(v.begin(),v.end(),2)-v.begin();//返回下标
注意：：set 不使用 STL 提供的 lower_Bound 函数！！

//permutation 排列
int a[5]={1,2,3,4,5};
 next_permutation(a,a+5);

//string 类字符串操作
while(ss.find(s[i])!=ss.npos){
            int now=ss.find(s[i]);
            num++;
            ss.erase(now,s[i].size());
            ss.insert(now,"-=-");
        }
```

9.bitset 操作

```
std::bitset<1000> bs; //声明

bitset(unsigned long val): //设为 val 的二进制形式。

bitset(const string& str): //设为   串 str。

count(): 返回 1 的数量。

flip(): 翻转每一位。

to_string(): 返回转换成的字符串表达。

To_ulong()转为 int
```

```
}
```

10. 对拍器

1. 数据生成代码 data.cpp 示例：

```
#include <bits/stdc++.h>
int main()
{
    struct _timeb T;
    _ftime(&T);
    srand(T.millitm);

    freopen("in.txt", "w", stdout); //生成 使两份基本代码 将要读入的数据
    int a = rand(), b = rand();
    printf("%d %d\n", a, b);
}
```

2. 暴力代码 baoli.cpp 示例：

```
#include <bits/stdc++.h>
int main()
{
    freopen("in.txt", "r", stdin);      //读入数据生成器造出来的数据
    freopen("baoli.txt", "w", stdout); //输出答案
    int a, b, ans = 0;
    scanf("%d %d", &a, &b);
    for (int i = 1; i <= a; ++i)
        ans++;
    for (int i = 1; i <= b; ++i)
        ans++;
    printf("%d\n", ans);
}
```

3. 正解代码 std.cpp 示例：

```
#include <bits/stdc++.h>
int main()
{
    freopen("in.txt", "r", stdin);
    freopen("std.txt", "w", stdout);
    int a, b;
    scanf("%d %d", &a, &b);
    printf("%d\n", a + b);
}
```

4. 对拍代码 duipai.cpp 示例：

```
#include<bits/stdc++.h>
```

```
using namespace std;
int main()
{
    while (1) //一直循环，直到找到不一样的数据
    {
        system("data.exe");
        system("baoli.exe");
        system("std.exe");
        if (system("fc std.txt baoli.txt")) //当 fc 返回 1 时，说明这时数据不一样
            break;                          //不一样就跳出循环
    }
    return 0;
}
```

5.运行对拍程序

目前，我们有了 4 份代码。为了实现对拍，我们要把这些代码放在同一个文件夹的同一层里。再次确保打开每一份代码，编译，让每一份代码都生成一个同名的 .exe 程序。