¼

1 数据结构

1.0.1 st 表

维护区间可重复贡献问题（如 gcd，min,max，&，｜等）

code

```
  int n;
  cin >> n;
  int log = log2(n);
  vector<vector<int>>st(n + 1, vector<int>(log + 1));
  for (int i = 1; i <= n; i++)cin >> st[i][0];
  vector<int>logn(n + 1);
  logn[1] = 0; logn[2] = 1;
  for (int i = 3; i <= n; i++)logn[i] = logn[i / 2] + 1;
  for (int j = 1; j <= log; j++) {
      for (int i = 1; i + (1 << j) - 1 <= n; i++) {
          st[i][j] = min(st[i][j - 1], st[i + (1 << (j - 1))][j - 1]);
      }
  }
int query(int l, int r, const vector<vector<int>>& st, const vector<int>&logn) {
    int m = logn[r - l + 1];
    return min(st[l][m], st[r - (1 << m) + 1][m]);
}
```

1.0.2 并查集

扁平化 + 递归

```
int fa[N+1];
int find(int x){
    if(x!=fa[x]){
        fa[x]=find(fa[x]);
    }
    return fa[x];//此处不能改为 x 因为此处是逐层返回
}
bool issame(int x,int y){
    return find(x)==find(y);
}
void un(int x,int y){
   fa[find(x)]=find(y);
}
void build(int x){
    for(int i=1;i<=x;i++){
        fa[i]=i;
    }
}
```

1.0.3 单调栈

1. 每个位置都求当前位置左侧和右侧比当前位置小，且距离最近的位置在哪（栈为大压小），最底元素是最小的

2. 或每个位置都求当前位置左侧和右侧比当前位置大，且距离最近的位置在哪（栈为小压大）

下面以情况一说明：

code:

```
stack<int>t;
int pre[n];//储存左侧数
int last[n];//储存右侧数
int a[n];
// pre/last=-1 表示不存在
    for(i=0;i<n;i++){
        cin>>a[i];
    }
t.push(0);
for(i=1;i<n;i++){
if(a[i]>a[t.top()]){
   t.push(i);
}
else{
while(!t.empty()&&a[i]<=a[t.top()])
{temp=t.top();
t.pop();
if(!t.empty()){
    pre[temp]=t.top();
}
else pre[temp]=-1;
last[temp]=i;
}
t.push(i);
    }
}
//如果没有重复数字 左右侧都是严格小于 不需要考虑下面代码
//如果有 如果不加下面代码 左侧是小于右侧是小于等于
//      加了的话是严格小于 应根据题目要求选择
while(!t.empty()){
    temp=t.top();
    t.pop();
    if(!t.empty()){
        pre[temp]=t.top();
    }
    else pre[temp]=-1;
    last[temp]=-1;
}
```

补充

如果两侧都不要严格递增或者递减，上面那个代码只能保证一侧

```
stack<int>st;
    st.push(1);
    for (int i = 2; i <= n; i++) {
        while (!st.empty() && a[i] >= a[st.top()]) {
            int temp = st.top();
            st.pop();
            if (l[temp] == 0) {
                if (st.empty())l[temp] = 0;
                else l[temp] = st.top();
            }
            r[temp] = i;
            if (a[i] == a[temp] && l[i] == 0)l[i] = temp;
        }
        st.push(i);
    }
    while (!st.empty()) {
        int temp = st.top();
        st.pop();
        if (l[temp] == 0) {
            if (st.empty())l[temp] = 0;
            else l[temp] = st.top();
        }
         r[temp] = n + 1;
    }
```

1.0.4 单调队列

头部维持最值优势，尾部维持新旧优势，新元素一定有新旧优势，又新最值又好可以直接淘汰前面又旧最值又差的，扩展时又淘汰了头部过期的元素

1.0.5 树状数组

注意下标要从 1 开始

1.0.5.1 一维

```
int lowbit(int x){return x&-x;}
void addtree(int x,int ad){
    while(x<=n){
        tr[x]+=ad;
        x+=lowbit(x);
    }
}
int querytree(int x){
    int an=0;
    while(x){
        an+=tr[x];
        x-=lowbit(x);
    }
    return an;
}
int query(int l,int r){
    return querytree(r)-querytree(l-1);
}
```

1.0.5.1.1 单点修改区间查询

```
int lowbit(int x){return x&-x;}
void addtree(int x,int ad){
    while(x<=n){
        tr[x]+=ad;
        x+=lowbit(x);
    }
}
int querytree(int x){
    int an=0;
    while(x){
        an+=tr[x];
        x-=lowbit(x);
    }
    return an;
}
void add(int l,int r,int ad){
    addtree(l,ad);
    addtree(r+1,-ad);
}

//main
for(int i=1;i<=n;i++)cin>>a[i];
    for(int i=1;i<=n;i++){
        addtree(i,a[i]-a[i-1]);
```

```
    }
```

1.0.5.1.2 区间修改单点查询

我们可以基于第二题的**差分**思想，考虑一下如何在问题二构建的树状数组上求前缀和：（在这里$a[i]$表示原数组，$b[i]$表示前缀和数组）求位置y的前缀和，我们可以推导出这样一个式子：$\Sigma_{i=1}^{y} a[i] = \Sigma_{i=1}^{y} \Sigma_{j=1}^{i} b[j] = \Sigma_{i=1}^{y} d[i] \times (y - i + 1) = (y + 1)\Sigma_{i=1}^{y} d[i] - \Sigma_{i=1}^{y} d[i] \times i$。

这样我们只需要维护两个数组的前缀和：一个是：$tree[i]\Sigma_{i=1}^{y} d[i]$，另一个是：$tree1[i] = \Sigma_{i=1}^{y} d[i] \times i$。

Figure 1: image-20250819105824797

```
int tr[N+1];
int tr1[N+1];
int lowbit(int x){return x&-x;}
void addtree(int x,int ad){
    for(int i=x;i<=n;i+=lowbit(i)){
        tr[i]+=ad;
        tr1[i]+=x*ad;
    }
}
int querytree(int x){
    int an=0;
    for(int i=x;i>0;i-=lowebit(i)){
        an+=(x+1)*tr[i]-tr1[i];
    }
    return an;
}
void add(int l,int r,int ad){
    addtree(l,ad);
    addtree(r+1,-ad);
}
int query(int l,int r){
    return querytree(r)-query(l-1);
}
```

1.0.5.1.3 区间修改区间查询

1.0.5.2 二维

```
ll tr[N+1][M+1];

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

void addtree(int x, int y, ll ad){
    for(int i=x;i<=n;i+=lowbit(i))
        for(int j=y;j<=m;j+=lowbit(j))
            tr[i][j]+=ad;
}
ll querytree(int x, int y){
    ll an=0;
    for(int i=x;i>0;i-=lowbit(i))
        for(int j=y;j>0;j-=lowbit(j))
            an+=tr[i][j];
    return an;
}

// 矩形和 (x1,y1)..(x2,y2)
ll query(int x1,int y1,int x2,int y2){
    return  querytree(x2,y2)
          - querytree(x1-1,y2)
          - querytree(x2,y1-1)
          + querytree(x1-1,y1-1);
}
```

1.0.5.2.1 单点修改区间查询

1.0.5.2.2 区间修改单点查询 此时修改是根据二维差分，修改 (x,y) 的值相当于修改 (x,y)->(n,m) 这个矩形范围内的值

```
ll tr[N+1][M+1];

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

void addtree(int x, int y, long long ad){
    for(int i=x;i<=n;i+=lowbit(i))
        for(int j=y;j<=m;j+=lowbit(j))
            tr[i][j]+=ad;
}
ll querytree(int x, int y){
    ll an=0;
    for(int i=x;i>0;i-=lowbit(i))
        for(int j=y;j>0;j-=lowbit(j))
            an+=tr[i][j];
    return an;
```

```
}
void add(int x1,int y1,int x2,int y2,long long ad){
    addtree(x1, y1, ad);
    if(y2+1<=m) addtree(x1, y2+1, -ad);
    if(x2+1<=n) addtree(x2+1, y1, -ad);
    if(x2+1<=n && y2+1<=m) addtree(x2+1, y2+1, ad);
}
```

```
ll tr1[N+2][M+2]; // 存 ad
ll tr2[N+2][M+2]; // 存 y*ad
ll tr3[N+2][M+2]; // 存 x*ad
ll tr4[N+2][M+2]; // 存 x*y*ad

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

void addtree(int x, int y, ll ad){
    for(int i=x;i<=n;i+=lowbit(i)){
        for(int j=y;j<=m;j+=lowbit(j)){
            tr1[i][j] += ad;
            tr2[i][j] += 1ll*y*ad;
            tr3[i][j] += 1ll*x*ad;
            tr4[i][j] += 1ll*x*y*ad;
        }
    }
}

// 矩形 [x1..x2][y1..y2] 加 ad
void add(int x1, int y1, int x2, int y2, ll ad){
    addtree(x1, y1, ad);
    if(x2+1<=n)     addtree(x2+1, y1, -ad);
    if(y2+1<=m)     addtree(x1, y2+1, -ad);
    if(x2+1<=n && y2+1<=m) addtree(x2+1, y2+1, ad);
}

// 前缀 (1..x,1..y) 的总和
ll querytree(int x, int y){
    ll an = 0;
    for(int i=x;i>0;i-=lowbit(i)){
        for(int j=y;j>0;j-=lowbit(j)){
            an += 1ll*(x+1)*(y+1)*tr1[i][j]
                - 1ll*(x+1)*tr2[i][j]
                - 1ll*(y+1)*tr3[i][j]
                + tr4[i][j];
        }
    }
```

```
    return an;
}

// 矩形和 (x1,y1)..(x2,y2)
ll query(int x1, int y1, int x2, int y2){
    return  querytree(x2, y2)
          - querytree(x1-1, y2)
          - querytree(x2, y1-1)
          + querytree(x1-1, y1-1);
}
```

1.0.5.2.3 区间修改区间查询

1.0.6 线段树

```
struct tree
{
    int l,r,ad,sum;
}tr[4*N+1];
void up(int i){
    tr[i].sum=tr[i<<1].sum+tr[i<<1|1].sum;
}
void lazy(int i,int ad){
    tr[i].ad+=ad;
    tr[i].sum+=(tr[i].r-tr[i].l+1)*ad;
}
void down(int i){
    if(tr[i].l!=tr[i].r&&tr[i].ad!=0){
        lazy(i<<1,tr[i].ad);
        lazy(i<<1|1,tr[i].ad);
        tr[i].ad=0;
    }
}
void build(int i,int l,int r){
    tr[i].l=l;
    tr[i].r=r;
    if(l==r){
        tr[i].sum=a[l];
        return;
    }
    int m=(l+r)>>1;
```

```
    build(i<<1,l,m);
    build(i<<1|1,m+1,r);
    up(i);
}
void add(int i,int ql,int qr,int ad){
    if(tr[i].l>=ql&&tr[i].r<=qr){
        lazy(i,ad);
        return;
    }
    if(qr<tr[i].l||ql>tr[i].r)return;
    down(i);
    int m=(tr[i].l+tr[i].r)>>1;
    if(ql<=m)add(i<<1,ql,qr,ad);
    if(qr>=m+1)add(i<<1|1,ql,qr,ad);
    up(i);
}
int query(int i,int ql,int qr){
    int an=0;
    if(tr[i].l>=ql&&tr[i].r<=qr){
        return tr[i].sum;
    }
    if (qr < tr[i].l || ql > tr[i].r) return 0;
    down(i);
    int m=(tr[i].l+tr[i].r)>>1;
    if(ql<=m)an+=query(i<<1,ql,qr);
    if(qr>=m+1)an+=query(i<<1|1,ql,qr);
    return an;
}
```

1.0.7 线段树解决区间合并

经典模型：

连续 1 的子串长度：给你一个 01 串，再给你多个区间，每次查询这个区间内最长的全为 1 的子串

此时线段树需要维护的信息：连续 1 的最长子串长度 len，连续 1 的最长前缀长度 pre，连续 1 的最长后缀长度 suf

$len[i] = max(len[i << 1], len[i << 1|1], suf[i << 1] + pre[i << 1|1])$

如果 i«1 全为 1，$pre[i] = pre[i << 1] + pre[i << 1|1]$，否则 $pre[i] = pre[i << 1]$

同理 如果 i«1|1 全为 1，$suf[i] = suf[i << 1] + suf[i << 1|1]$，否则 $suf[i] = suf[i << 1|1]$

```
struct tree
{
    int l,r,ad,pre,suf,len;
}tr[4*N+1];
struct node{
int  pre,len,suf;
};
void up(int i){
    tr[i].len=max({tr[i<<1].len,tr[i<<1|1].len,tr[i<<1].suf+tr[i<<1|1].pre});
    tr[i].pre=(tr[i<<1].len==tr[i<<1].r-
     ↪ tr[i<<1].l+1)?tr[i<<1].pre+tr[i<<1|1].pre:tr[i<<1].pre;
    tr[i].suf=(tr[i<<1|1].len==tr[i<<1|1].r-
     ↪ tr[i<<1|1].l+1)?tr[i<<1].suf+tr[i<<1|1].suf:tr[i<<1|1].suf;
}
void lazy(int i,int ad){
    tr[i].ad=ad;
    tr[i].len=tr[i].pre=tr[i].suf=(ad==1?tr[i].r-tr[i].l+1:0);
}
void down(int i){
    if(tr[i].l!=tr[i].r&&tr[i].ad!=0){
        lazy(i<<1,tr[i].ad);
        lazy(i<<1|1,tr[i].ad);
        tr[i].ad=0;
    }
}
void build(int i,int l,int r){
    tr[i].l=l;
    tr[i].r=r;
    if(l==r){
        tr[i].pre=tr[i].suf=tr[i].len=1;
        return;
    }
    int m=(l+r)>>1;
    build(i<<1,l,m);
    build(i<<1|1,m+1,r);
    up(i);
}
void add(int i,int ql,int qr,int ad){
    if(tr[i].l>=ql&&tr[i].r<=qr){
        tr[i].ad=ad;
        tr[i].len=tr[i].pre=tr[i].suf=(ad==1?tr[i].r-tr[i].l+1:0);
        return;
    }
    if(qr<tr[i].l||ql>tr[i].r)return;
```

```
    down(i);
    int m=(tr[i].l+tr[i].r)>>1;
    if(ql<=m)add(i<<1,ql,qr,ad);
    if(qr>=m+1)add(i<<1|1,ql,qr,ad);
    up(i);
}
node query(int i,int ql,int qr){
    if(tr[i].l>=ql&&tr[i].r<=qr){
        return {tr[i].pre,tr[i].len,tr[i].suf};
    }
    down(i);
    int m=(tr[i].l+tr[i].r)>>1;
    if(qr<=m) return query(i<<1,ql,qr);
    else if(ql>=m+1) return query(i<<1|1,ql,qr);
    else{
        node x=query(i<<1,ql,qr),y=query(i<<1|1,ql,qr);
        node z;
        z.len=max({x.len,y.len,x.suf+y.pre});
        z.pre=tr[i<<1].r-max(ql,tr[i<<1].l)+1==x.pre?x.pre+y.pre:x.pre;
        z.suf=min(qr,tr[i<<1|1].r)-tr[i<<1|1].l+1==y.suf?x.suf+y.suf:y.suf;
        return z;
    }
}
```

1.0.8 线段树的最值操作和区间修改和历史最值

给你一个 arr，支持一下三种操作

1 l ， r ， x，把区间 arr[l,r] 的数都更新为 $min(x,a[i])$

2 l r 查询区间的最大值

3 l r 查询区间的累加和

4 l r x 把范围 l~r 都 +x

此时线段树需要维护的信息有 累加和 sum，区间最大值 maxn，区间最大值的个数 cnt，区间次大值 sema（严格小于最大值），如果没有设置为负无穷

对于取 min 操作

如果区间 maxn<=x，那么直接返回

如果 x>sema&&x<=maxn，可以懒更新，maxn=x,sum-=cnt*(maxn-x)

如果 x<=sema 那么直接暴力向下递归

时间复杂度是 $O(mlogn)$ 的

证明：对于线段树的每个节点，如果它的 maxn!= 它的父亲的 maxn，那么给这个点打一个标记，标记的数量是 $O(n)$ 的，每次 setmin 经过的点的标记都删除，并且没有标记的点之后不会再访问了，每次删除标记的复杂度最多是 $O(logn)$= 树高

对于操作 4：

此时势能增加量不会超过 $(logn)^2$，因为修改区间，只会给从 1 到 tr[i].l=tr[i].r=l 所在节点以及 1 到 r 所在节点打上标记，标记个数为 $2*logn$，又删除每个标记时间复杂度最差是 $O(logn)$

历史最值

除了一个长度为 n 的数组 a，还给了一个辅助数组 b=a，每次操作完，令 $bi=max(bi,ai)$

此时还要额外储存三个信息，maxhistory(历史的最大值)，maxtop(最大值的历史最大 maxadd)，othtop(除了最大值的历史最大 maxoth)

```
struct tree{
    int l,r,maxn,maxhis,maxadd,othadd,maxaddtop,othaddtop,cntmax,sema;
    ll sum;
}tr[(N<<2)+1];
void up(int i){
    tr[i].sum=tr[i<<1].sum+tr[i<<1|1].sum;
    tr[i].maxhis=max(tr[i<<1].maxhis,tr[i<<1|1].maxhis);
    tr[i].maxn=max(tr[i<<1].maxn,tr[i<<1|1].maxn);
    if(tr[i<<1].maxn>tr[i<<1|1].maxn){
        tr[i].cntmax=tr[i<<1].cntmax;
        tr[i].sema=max(tr[i<<1].sema,tr[i<<1|1].maxn);
    }
    else if(tr[i<<1].maxn<tr[i<<1|1].maxn){
        tr[i].cntmax=tr[i<<1|1].cntmax;
        tr[i].sema=max(tr[i<<1|1].sema,tr[i<<1].maxn);
    }
    else{
        tr[i].cntmax=tr[i<<1].cntmax+tr[i<<1|1].cntmax;
        tr[i].sema=max(tr[i<<1].sema,tr[i<<1|1].sema);
    }
}
void lazy(int i,int maxadd,int othadd,int maxaddtop,int othaddtop){
    tr[i].maxhis=max(tr[i].maxhis,tr[i].maxn+maxaddtop);
    tr[i].maxaddtop=max(tr[i].maxaddtop,tr[i].maxadd+maxaddtop);
    tr[i].othaddtop=max(tr[i].othaddtop,tr[i].othadd+othaddtop);
    tr[i].maxadd+=maxadd;
    tr[i].othadd+=othadd;
    tr[i].sum+=1ll*maxadd*tr[i].cntmax+1ll*othadd*(tr[i].r-tr[i].l+1-tr[i].cntmax);
    tr[i].maxn+=maxadd;
```

```
    tr[i].sema+=tr[i].sema==-inf?0:othadd;
}
void down(int i){
    int maxn=max(tr[i<<1].maxn,tr[i<<1|1].maxn);
    if(tr[i<<1].maxn==maxn)
        lazy(i<<1,tr[i].maxadd,tr[i].othadd,tr[i].maxaddtop,tr[i].othaddtop);
    else lazy(i<<1,tr[i].othadd,tr[i].othadd,tr[i].othaddtop,tr[i].othaddtop);
    if(tr[i<<1|1].maxn==maxn)
        lazy(i<<1|1,tr[i].maxadd,tr[i].othadd,tr[i].maxaddtop,tr[i].othaddtop);
    else lazy(i<<1|1,tr[i].othadd,tr[i].othadd,tr[i].othaddtop,tr[i].othaddtop);
    tr[i].maxadd=tr[i].othadd=tr[i].maxaddtop=tr[i].othaddtop=0;
}
void build(int i,int l,int r){
    tr[i].l=l;
    tr[i].r=r;
    if(l==r){
        tr[i].sum=tr[i].maxn=tr[i].maxhis=a[l];
        tr[i].sema=-inf;
        tr[i].cntmax=1;
        return;
    }
    int m=(l+r)>>1;
    build(i<<1,l,m);
    build(i<<1|1,m+1,r);
    up(i);
}
void add(int i,int l,int r,int ad){
    if(tr[i].l>=l&&tr[i].r<=r){
        lazy(i,ad,ad,ad,ad);
        return;
    }
    if(r<tr[i].l||l>tr[i].r)return;
    down(i);
    int m=(tr[i].l+tr[i].r)>>1;
    if(l<=m)add(i<<1,l,r,ad);
    if(r>=m+1)add(i<<1|1,l,r,ad);
    up(i);
}
void setmin(int i,int l,int r,int x){
    if(tr[i].maxn<=x)return;
    if(tr[i].l>=l&&tr[i].r<=r&&tr[i].sema<x){
        lazy(i,x-tr[i].maxn,0,x-tr[i].maxn,0);
        return;
    }
    down(i);
    int m=(tr[i].l+tr[i].r)>>1;
    if(l<=m)setmin(i<<1,l,r,x);
```

```
    if(r>=m+1)setmin(i<<1|1,l,r,x);
    up(i);
}
ll querysum(int i,int l,int r){
    if(tr[i].l>=l&&tr[i].r<=r){
        return tr[i].sum;
    }
    if(r<tr[i].l||l>tr[i].r)return 0;
    down(i);
    ll an=0;
    int m=(tr[i].l+tr[i].r)>>1;
    if(l<=m)an+=querysum(i<<1,l,r);
    if(r>=m+1)an+=querysum(i<<1|1,l,r);
    return an;
}
int querymax(int i,int l,int r){
    if(tr[i].l>=l&&tr[i].r<=r){
        return tr[i].maxn;
    }
    if(r<tr[i].l||l>tr[i].r)return -inf;
    down(i);
    int an=-inf;
    int m=(tr[i].l+tr[i].r)>>1;
    if(l<=m)an=max(an,querymax(i<<1,l,r));
    if(r>=m+1)an=max(an,querymax(i<<1|1,l,r));
    return an;
}
int querymaxhis(int i,int l,int r){
    if(tr[i].l>=l&&tr[i].r<=r){
        return tr[i].maxhis;
    }
    if(r<tr[i].l||l>tr[i].r)return -inf;
    down(i);
    int an=-inf;
    int m=(tr[i].l+tr[i].r)>>1;
    if(l<=m)an=max(an,querymaxhis(i<<1,l,r));
    if(r>=m+1)an=max(an,querymaxhis(i<<1|1,l,r));
    return an;
}
```

1.0.9 动态开点线段树

适用的范围：需要支持的范围很大但是实际操作的次数不大

此时使用的空间约为 $2mlogn$，适当调大即可

可以支持很大的范围，一开始不为每个范围都分配空间

当真的需要开辟左侧右侧的空间时，再临时申请

```
struct tree{
    int l,r,ad,sum;
}tr[N+1];
int cnt=1;
void up(int i,int l,int r){
    tr[i].sum=tr[l].sum+tr[r].sum;
}
void lazy(int i,int len,int ad){
    tr[i].ad+=ad;
    tr[i].sum+=len*ad;
}
void down(int i,int llen,int rlen){
    if(tr[i].ad!=0){
        if(!tr[i].l){
            tr[i].l=++cnt;
        }
        if(!tr[i].r){
            tr[i].r=++cnt;
        }
        lazy(tr[i].l,llen,tr[i].ad);
        lazy(tr[i].r,rlen,tr[i].ad);
        tr[i].ad=0;
    }
}
void add(int i,int l,int r,int ql,int qr,int ad){
    if(ql<=l&&qr>=r){
        lazy(i,r-l+1,ad);
        return;
    }
    int m=(l+r)>>1;
    down(i,m-l+1,r-m);
    if(ql<=m){
        if(!tr[i].l)tr[i].l=++cnt;
        add(tr[i].l,l,m,ql,qr,ad);
    }
    if(qr>=m+1){
        if(!tr[i].r)tr[i].r=++cnt;
        add(tr[i].r,m+1,r,ql,qr,ad);
```

```
    }
    up(i,tr[i].l,tr[i].r);
}
int query(int i,int l,int r,int ql,int qr){
    if(ql<=l&&qr>=r)return tr[i].sum;
    if(qr<l||ql>r)return 0;
    int an=0,m=(l+r)>>1;
    down(i,m-l+1,r-m);
    if(ql<=m&&tr[i].l)an+=query(tr[i].l,l,m,ql,qr);
    if(qr>=m+1&&tr[i].r)an+=query(tr[i].r,m+1,r,ql,qr);
    return an;
}

//main
    if(op==1){
    int l,r,ad;
    cin>>l>>r>>ad;
    add(1,1,n,l,r,ad);
    }
    else{
    int l,r;
    cin>>l>>r;
    cout<<query(1,1,n,l,r)<<'\n';
    }
```

1.0.10 可持久化线段树

1.0.10.1 单点修改单点查询 root[i] 存的是第 i 号版本的头结点，每次如果要修改的话 因为是单点修改，只会改变一条链 $O(logn)$ 的点，此时新生成一个节点，把所有信息复用之前的节点，然后再去修改

```
const int N=1e6;
int a[N+1];
int root[23*N+1];
struct tree{
    int l,r,val;
}tr[23*N+1];
int trcnt;
int build(int l,int r){
    int cur=++trcnt;
    tr[cur].l=l,tr[cur].r=r;
    if(l==r){
        tr[cur].val=a[l];
        return cur;
    }
    int m=(l+r)>>1;
    tr[cur].l=build(l,m);
    tr[cur].r=build(m+1,r);
    return cur;
}
int add(int x,int ad,int l,int r,int fa){
    int cur=++trcnt;
    tr[cur].l=tr[fa].l;
    tr[cur].r=tr[fa].r;
    tr[cur].val=tr[fa].val;
    if(l==r){
        tr[cur].val=ad;
        return cur;
    }
    int m=(l+r)>>1;
    if(x<=m)tr[cur].l=add(x,ad,l,m,tr[cur].l);
    else tr[cur].r=add(x,ad,m+1,r,tr[cur].r);
    return cur;
}
int query(int x,int l,int r,int cur){
    if(l==r)return tr[cur].val;
    int m=(l+r)>>1;
    if(x<=m)return query(x,l,m,tr[cur].l);
    else return query(x,m+1,r,tr[cur].r);
}
//main
 root[0]=build(1,n);
    for(int i=1;i<=q;i++){
        int v,op,p;
        cin>>v>>op>>p;
        if(op==1){
            int ad;
            cin>>ad;
```

```
            root[i]=add(p,ad,1,n,root[v]);
        }
        else{
            root[i]=root[v];
            cout<<query(p,1,n,root[i])<<'\n';
        }
    }
```

1.0.10.2 单点修改范围查询 打印 l-r 第 k 小的数

先将值域离散化得到 sorted 数组，线段树有 $sorted.size()$ 个节点，每个节点存储的是=sorted[i] 的点的个数，再建立 n 个版本，第 i 个版本存储的是原数组下标 1-i 范围内的 sorted 数组的个数

对于查询 l,r,k 我们需要版本 l-1 和版本 r，到达一个节点此时值域的范围为 ql-qr，如果 $r:num[ql-qm]-\ l-1:num[ql-qm]>k$ 那说明要找的点在左侧，否则要找的点在右侧并且 k-=diff(刚才 num 的差值)

```
const int N=2e5;
vector<int>sorted;
int a[N+1];
int root[N+1];
struct tree{
    int l,r,siz;
}tr[23*N+1];
int trcnt;
int build(int l,int r){
    int cur=++trcnt;
    if(l==r);
    else{
        int m=(l+r)>>1;
        tr[cur].l=build(l,m);
        tr[cur].r=build(m+1,r);
    }
    return cur;
}
int add(int x,int ad,int l,int r,int fa){
    int cur=++trcnt;
    tr[cur].l=tr[fa].l;
    tr[cur].r=tr[fa].r;
    tr[cur].siz=tr[fa].siz+1;
    if(l==r);
    else{
        int m=(l+r)>>1;
        if(x<=m)tr[cur].l=add(x,ad,l,m,tr[cur].l);
        else tr[cur].r=add(x,ad,m+1,r,tr[cur].r);
```

```
    }
    return cur;
}
int query(int k,int cur1,int cur2,int l,int r){
    if(l==r)return l;
    int diff=tr[tr[cur2].l].siz-tr[tr[cur1].l].siz;
    int m=(l+r)>>1;
    if(diff>=k){
        return query(k,tr[cur1].l,tr[cur2].l,l,m);
    }
    else{
        return query(k-diff,tr[cur1].r,tr[cur2].r,m+1,r);
    }
}

//main
 for(int i=1;i<=n;i++){
        int x;
        cin>>x;
        a[i]=x;
        sorted.push_back(x);
    }
    sort(all(sorted));
    sorted.erase(unique(all(sorted)),sorted.end());
    int m=sorted.size();
    sorted.insert(sorted.begin(),-1);
    root[0]=build(1,m);
    for(int i=1;i<=n;i++){
        int num=lower_bound(all(sorted),a[i])-sorted.begin();
        root[i]=add(num,1,1,m,root[i-1]);
    }
    while(q--){
        int l,r,k;
        cin>>l>>r>>k;
        cout<<sorted[query(k,root[l-1],root[r],1,m)]<<'\n';
    }
```

给一个长度为 n 的序列，每次给 l,r，将 l->r 范围内每个值第一次出现的位置拿出来，求 len/2 位置的值

从右向左遍历原数组建不同版本的树，线段树长度和原数组等长，如果 i 是此时版本最左的 y=a[i] 的位置，那么线段树在 i 位置 +1，在上一个位置-1，此时查询 l-r，就查询 l 版本 1-r 或者 l-r 范围内 1 的个数，再在树上找到精确位置，如果左孩子的 sum>=k 那么往左找，否则往右并且把 sum-k

给一个长度为 n 的序列，每次给 l,r，求 l-r 范围内权值的 mex

从右向左建立版本，维护值域线段树，点 i 的值表示 val=i 的最左下标，区间如果这个

范围有最左为 0 那么为 0，否则取子节点的 max，对于一个查询，利用 1 版本的在树上找，如果左树的值 >r 那么往左找，否则往右找

还是类似求 l-r 第 k 大，此时 dfn[u] 版本要从 dfn[u]-1 推过来，线段树位置为 k 的地方维护此时树高为 k 的子树的大小之和

1.0.10.3 范围修改 经典方法实现

需要懒更新机制但是仿照单点修改的可持久化线段树，来到一个节点的时候都新建节点并且 clone 出原来那个节点的信息，如果此时懒信息要下发，那么左右孩子都要新建来接收懒信息以保证老节点的信息不改变

建立最初线段树空间 $O(4N)$，建立 n 个版本的线段树空间 $O(nlogn)$,m 次查询空间 $O(mlogn)$

```
int root[100*N+1];
struct tree{
    int l,r,ad;
    ll sum;
}tr[100*N+1];
int trcnt;
void up(int i){
    tr[i].sum=tr[tr[i].l].sum+tr[tr[i].r].sum;
}
int clone(int i){
    int cur=++trcnt;
    tr[cur].sum=tr[i].sum;
    tr[cur].l=tr[i].l;
    tr[cur].r=tr[i].r;
    tr[cur].ad=tr[i].ad;
    return cur;
}
void lazy(int cur,int len,int ad){
    tr[cur].sum+=1ll*len*ad;
    tr[cur].ad+=ad;
}
void down(int i,int llen,int rlen){
    if(tr[i].ad!=0){
        tr[i].l=clone(tr[i].l);
        tr[i].r=clone(tr[i].r);
        lazy(tr[i].l,llen,tr[i].ad);
        lazy(tr[i].r,rlen,tr[i].ad);
        tr[i].ad=0;
    }
}
int build(int l,int r){
    int cur=++trcnt;
    if(l==r)tr[cur].sum=a[l];
```

```
    else{
        int m=(l+r)>>1;
        tr[cur].l=build(l,m);
        tr[cur].r=build(m+1,r);
        up(cur);
    }
    return cur;
}
int add(int jl,int jr,int l,int r,int ad,int fa){
    int cur=clone(fa);
    if(jl<=l&&jr>=r){
        lazy(cur,r-l+1,ad);
    }
    else{
        int m=(l+r)>>1;
        down(cur,m-l+1,r-m);
        if(jl<=m)tr[cur].l=add(jl,jr,l,m,ad,tr[cur].l);
        if(jr>=m+1)tr[cur].r=add(jl,jr,m+1,r,ad,tr[cur].r);
        up(cur);
    }
    return cur;
}
ll query(int jl,int jr,int l,int r,int cur){
    if(jl<=l&&jr>=r){
        return tr[cur].sum;
    }
    int m=(l+r)>>1;
    down(cur,m-l+1,r-m);
    ll an=0;
    if(jl<=m)an+=query(jl,jr,l,m,tr[cur].l);
    if(jr>=m+1)an+=query(jl,jr,m+1,r,tr[cur].r);
    return an;
}
```

1.0.10.3.1 标记永久化 比起经典方法更节省空间

但只适用于修改和查询都有可叠加性（如范围加一个数和求范围 sum）

不适用于如范围重置和查询最大值和最小值

此时懒更新信息不再往下发，修改除非 $jl <= l\&\&jr >= r$ 才修改 ad 信息，其他的直接修改 sum

对于查询除非 $jl <= l\&\&jr >= r$ 才加 sum 信息，其他的只增加 ad 信息

此时的空间消耗变为 $O(4n + nlogn)$（因为没了懒更新机制下发的新建节点）

```
int a[N+1];
int root[N+1];
struct tree{
    int l,r,ad;
    ll sum;
}tr[30*N+1];
int trcnt;
int build(int l,int r){
    int cur=++trcnt;
    if(l==r)tr[cur].sum=a[l];
    else{
        int m=(l+r)>>1;
        tr[cur].l=build(l,m);
        tr[cur].r=build(m+1,r);
        tr[cur].sum=tr[tr[cur].l].sum+tr[tr[cur].r].sum;
    }
    return cur;
}
int add(int jl,int jr,int ad,int l,int r,int fa){
    int cur=++trcnt;
    tr[cur].l=tr[fa].l;
    tr[cur].r=tr[fa].r;
    tr[cur].ad=tr[fa].ad;
    tr[cur].sum=tr[fa].sum;
    tr[cur].sum+=1ll*ad*(min(r,jr)-max(l,jl)+1);
    if(jl<=l&&jr>=r){
        tr[cur].ad+=ad;
    }
    else{
        int m=(l+r)>>1;
        if(jl<=m)tr[cur].l=add(jl,jr,ad,l,m,tr[cur].l);
        if(jr>=m+1)tr[cur].r=add(jl,jr,ad,m+1,r,tr[cur].r);
    }
    return cur;
}
ll query(int jl,int jr,int l,int r,int cur){
    ll an=1ll*(min(jr,r)-max(jl,l)+1)*tr[cur].ad;
    if(jl<=l&&jr>=r){
        return tr[cur].sum;
    }
    int m=(l+r)>>1;
    if(jl<=m)an+=query(jl,jr,l,m,tr[cur].l);
    if(jr>=m+1)an+=query(jl,jr,m+1,r,tr[cur].r);
    return an;
}
```

2 dp

2.0.1 背包 dp

2.0.1.1 01 背包 压缩前

```
dp[i][j] 前 i 件物品背包容量 j 最大价值
    dp[i][j]=max(dp[i-1][j],dp[i-1][j-cost[i]]+value[i])
```

code:

```
const int n;
int dp[n+1];
int value[n+1];
int cost[n+1];
//n 为物品数 m 为背包大小
for(i=1;i<=n;i++){
    for(j=m;j>=cost[i];j--){
        dp[j]=max(dp[j],dp[j-cost[i]]+value[i]);
    }
}
return dp[m];
```

难题

思路 将总的石头分为接近 sum/2 的两堆（一堆大于等于一堆小于等于）用 01 背包（背包容量 sum/2）找到最多石头质量 m

re=sum-2*m

01 背包小拓展（物品的价值会随着背包容量变化）

由于这个状态转移方程 dp[j]=max(dp[j],dp[j-cost[i]]+value[i]);

在后面 dp 的东西是放在背包的最后侧

那么我们就要后 dp 受背包容量影响使答案增大幅度最大的

2.0.1.2 分组背包 看成每一组内的 01 背包

思路：循环组-> 循环背包容量-> 循环组内每一个物品

code:

```
//num 代表组的数量 cnt 代表该组内物品数量 m 代表背包容量
for(int k=1;k<=num;k++){
    for(i=m;i>=0;i--){
        for(j=1;j<=cnt[k];j++){
            if(i>=w[t[k][j]]){
                dp[i]=max(dp[i],dp[i-w[k][j]]+c[t[k][j]]);
            }
        }
    }
}
```

2.0.1.3 有依赖的背包 可以看成分组背包，以一个主件作为一个组，每一组里面包含只选主件和主件加附件

如果不用分组背包 ps：要开另外一个数组 在每一个组内 dp2 只能利用原来的 dp，操作完该组后再拷贝到 dp

2.0.1.4 完全背包 压缩前

```
dp[i][j] 前 i 件物品背包容量 j 最大价值
    dp[i][j]=max(dp[i-1][j],dp[i][j-cost[i]]+value[i])
                            //此处和 01 不同，01 是 dp[i-1][j-cost[i]]
```

与 01 相反

code

```
  for (int i = 0; i < m; i++) {
        for (int j = c[i]; j <= t; j++) {//注意 j 循环顺序不能变 不然就是 01 背包了
            dp[j] = max(dp[j], dp[j - c[i]] + v[i]);
        }
    }
```

2.0.1.5 多重背包 每个物品有多件，有 n 件物品可以看作选 n 次的 01 背包

二进制优化

二进制优化 将物品分作 2 的次方的大物品 大物品组合可以表示所有的数

code：

```
    for (int i = 1; i <= n; i++) {
        cin >> v >> w >> m;//v 价值 w 重量 m 个数
        ll c = 1;
        while (m > c) {
            m -= c;
            ve.push_back({ c * w,c * v });
            c <<= 1;
        }
        ve.push_back({ m * w,m * v });
    }
```

单调队列优化

根据物品重量的余数分组

```
//w 重量 t 背包容量
for(int i=0;i<w;i++){
    for(int j=i;j<c;j+=w){
    }
}
```

未空间压缩版本

```
//n 组物品 背包容量 t v 价值 w 重量 m 个数
ll n, t, v, w, m,temp;
ll ca(const vector<vector<ll>>&dp, int i, int k) {//不传引用会 tle
    return dp[i - 1][k] - k / w * v;
}
int main() {
    ios::sync_with_stdio(false);
    cin.tie(0), cout.tie(0);
    cin >> n >> t;
    vector<vector<ll>>dp(n + 1, vector<ll>(t + 1,0));
    for (int i = 1; i <= n; i++) {
        cin >> v >> w >> m;
        for (int j = 0; j <= min(t, w - 1); j++) {
            deque<ll>qu;//注意此时队列要写在第二层循环里面
            for (int k = j; k <= t; k += w) {
                while (!qu.empty()&&ca(dp,i,k)>=ca(dp,i,qu.back())) {
                    qu.pop_back();
                }
                qu.push_back(k);
                if (qu.front() == k - (m + 1) * w)qu.pop_front();
                temp = qu.front();
                dp[i][k] = k / w * v + ca(dp, i, temp);
            }
        }
```

```
    }
    cout << dp[n][t] << endl;
```

空间压缩版本

```
//即从右往左计算
ll w, v, m;
ll n, t;
int ca(vector<ll>& dp, int k) {
    return dp[k] - k / w * v;
}
    cin >> n >> t;
    vector<ll>dp(t + 1, 0);
    for (int i = 0; i < n; i++) {
        cin >> v >> w >> m;
        for (int j = 0; j <= min(t, w - 1); j++) {
            deque<int>de;
            for (int k = t - j, cnt = 1; k > 0 && cnt <= m; k -= w, cnt++) {
                while (!de.empty() && ca(dp, de.back()) <= ca(dp, k)) {
                    de.pop_back();
                }
                de.push_back(k);
            }
            for (int k = t - j, enter = k - m * w; k > 0; k -= w, enter -= w) {
                if (enter >= 0) {
                    while (!de.empty() && ca(dp, de.back()) <= ca(dp,
                     ↪  enter))de.pop_back();
                    de.push_back(enter);
                }
                dp[k] = k / w * v + ca(dp, de.front());
                if (de.front() == k)de.pop_front();
            }
        }
    }
    cout << dp[t] << endl;
```

2.0.2 期望 dp

期望DP

1. 顺推：dp[i]从哪来，分析所有dp[i+1]的情况
2. 逆推：dp[i]到哪去，分析所有dp[i]的情况

e.g.

1.

题目描述

有一个n面的骰子，每次等概率地投出任意一面，请问每个面都投出至少一次的期望次数是多少？

1.

dp[i]：投出了i个面的期望次数

dp[0]=0;

$$dp[i]=\frac{n-(i-1)}{n}[dp[i-1]+1]+\frac{i-1}{n}[dp[i-1]+1+dp[i]-dp[i-1]]$$

每次由i-1到i

有点类似前缀和

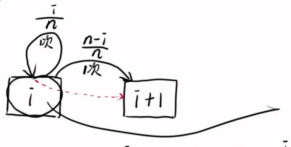
顺推：分析所有i-1到i的情况

1. 一次直接到
2. 先回到i-1再到i

2.

dp[i]：已经投出i个面，到投出n面所需期望次数

dp[n]=0



$$dp[i]=\frac{n-i}{n}\times(1+dp[i+1])+\frac{i}{n}(1+dp[i])$$

逆推：分析所有i的去向

前面两种方法都是将一个点的所有去向分析清楚了

如果不是这么考虑，则每个点都要多加一个P_i值

3.

$P[i]$：投出i个面的概率

$$P[i]=\underbrace{[P[i]\times\frac{i}{n}]}_{P_1}+\underbrace{[P[i-1]\times\frac{n-i+1}{n}]}_{P_2}$$

$$dp[i]=\frac{P_1}{P_1+P_2}(dp[i]+1)+\frac{P_2}{P_1+P_2}(dp[i-1]+1)$$

此时概率之和不一定为0，则分母需为P_1+P_2

e.g.

2.

小e最近迷上了"跳房子"的游戏，房子在一条长度为n的横轴上，1是起点，n是终点。

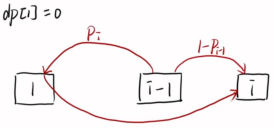
小e从起点开始，每次可以往右跳一格（即使得当前的坐标位置+1），但是由于每个房子上面的摩擦系数μ不同，每次从房子i起跳时，有p_i的概率因为脚滑回到起点（确实有点逆天），请问小e跳到终点的期望次数是多少？

最后的结果对10^9+7取模。

1. 正推

dp[i]：从1到i的期望次数

dp[1]=0



$$dp[i]=(1-P_{i-1})\times(\underbrace{dp[i-1]+1}_{\text{到}i})+P_{i-1}\times(\underbrace{dp[i-1]+1}_{\text{回到}i-1}+\underbrace{dp[i]}_{\text{到}i})$$

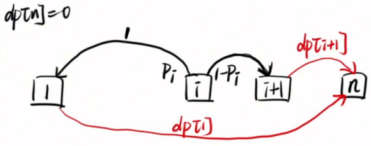
$$=dp[i-1]+1+P_{i-1}\times dp[i]$$

$$dp[i]=\frac{dp[i-1]+1}{1-P_{i-1}}$$

2. 逆

dp[i]：从i到n的期望步数

dp[n]=0



$$dp[i]=\underbrace{P_i(1+dp[i])}_{i\to n}+\underbrace{(1-P_i)\times(1+dp[i+1])}_{i\to i+1\to n}$$

P.S 此时逆推做法不可取，因为dp[i]会由dp[i]决定具有后效性。

2.0.3 换根 dp

树上某些问题需要知道以不同节点作为根的时候的答案

换根思考点：如果得到父节点作为根的答案，如何加工出来子节点作为根的答案

流程：

先以 1 节点作为根 dfs1 一遍收集一些信息，然后再从 1 节点开始进行树的遍历，求解每个节点作为根的时候的答案，进行 dfs2

```
void dfs(int u){
    dfn[u]=cnt++;
    for(auto v:g[u]){
        if(!dfn[v]){
            dfs(v);
            siz[u]+=siz[v];
        }
    }
}
void dfs2(int u){
    for(auto v:g[u]){
        if(dfn[v]>dfn[u]){
            an[v]=an[u]+n-2*siz[v];
            dfs2(v);
        }
    }
}
```

2.0.4 轮廓线 dp

轮廓线 dp 是去优化状压 dp

面对 n 行 m 列二维网格需要逐行逐列做决策并且状态信息表示上一行状态的时候，此时为了得到当前行的状态需要暴力 dfs

时间复杂度是 $O(2^m * n * 2^m)$

轮廓线 dp 的原理

使用轮廓线 dp $f(i,j,s)$ 表示当前来到 i 行 j 列，轮廓线的状况为 s

此时 $i-1$ 行 $j->m-1$ 列用 $s[j....m-1]$ 表示

i 行 $0->j-1$ 列用 $s[0...j-1]$ 表示

像一条轮廓线一样包裹

此时的时间复杂度是 $O(2^m * n * m)$

```
int sta[N];
const int mod=1e8;
int dp[N][M][1<<M];
int n,m;
int dfs(int i,int j,int s){
    if(i==n)return 1;
    if(dp[i][j][s]!=-1)return dp[i][j][s];
    ll an=dfs(j==m-1?i+1:i,j==m-1?0:j+1,s&(~(1<<j)));
    if(j==0&&sta[i]>>j&1&&!(s>>j&1))an=(an+dfs(j==m-1?i+1:i,j==m-
     ↪  1?0:j+1,s|(1<<j)))%mod;
    if(j>0&&sta[i]>>j&1&&!(s>>j&1)&&!(s>>(j-1)&1))an=(an+dfs(j==m-1?i+1:i,j==m-
     ↪  1?0:j+1,s|(1<<j)))%mod;
    dp[i][j][s]=an;
    return an;
}
```

2.0.5 区间 dp

区间 dp：大范围的问题拆分成若干小范围的问题来求解

可能性展开的常见方式：1）基于两侧端点讨论的可能性展开 2）基于范围上划分点的可能性展开

将小区间合并为大区间

区间长度由小到大 dp

```
   for(int len=1;len<=n;len++){
            for(int l=0;l+len-1<n;l++){
                int r=l+len-1;

            }
        }
```

2.0.6 树形 dp

树 头节点没有父亲，其他节点只有一个父亲的有向无环图，直观理解为发散状 在树上，从头节点出发到任何节点的路径是唯一的，不管二叉树还是多叉树都如此

树型 dp 在树上做动态规划，依赖关系比一般动态规划简单 因为绝大部分多数都是父依赖子 只是依赖关系简单，不代表题目简单

树型 dp 套路 1）分析父树得到答案需要子树的哪些信息 2）把子树信息的全集定义成递归返回值 3）通过递归让子树返回全集信息 4）整合子树的全集信息得到父树的全集信息并返回

2.0.6.1 树上背包 根据题目意思，每个点只有一个前继节点，则可以想到是树结构

code

```
int dfs(int u,int pre,int m){
    if(m==0)return 0;
    if(pre==0||m==1)return val[u];
    if(memo[u][pre][m]!=-1)return memo[u][pre][m];
    int an=0;
    int v=graph[u][pre-1];
    for(int i=0;i<m;i++){
        an=max(an,dfs(u,pre-1,m-i)+dfs(v,out[v],i));
    }
    memo[u][pre][m]=an;
    return an;
}
    memo.resize(n+1);
    for(int i=0;i<=n;i++)memo[i].resize(out[i]+1);
    for(int i=0;i<=n;i++){
        for(int j=0;j<=out[i];j++){
            memo[i][j].resize(m+2);
        }
    }
    for(int i=0;i<=n;i++){
        for(int j=0;j<=out[i];j++){
            for(int k=0;k<=m+1;k++){
                memo[i][j][k]=-1;
            }
        }
    }
    cout<<dfs(0,out[0],m+1);
```

2.0.7 数位 dp

前导 0 指将 1 补成 00001，对于某些问题前导 0 无影响如求所有数位之和，则可以不考虑前导 0

但对于如每个数位各不相同问题，10 本来符合题意，但补成 010 就不符合了，此时要考虑前导 0

前导 0

不管前导 0 就不需要 is_num 参数 不用考虑跳过本位直接填 0

ps：下面两个代码都是包括 0 的，如果不包括还需将 0 情况剔除

1. 不考虑前导 0

只有上界

code:

```
//枚举到 n
string s=to_string(n);
int m=sr.size();
int dfs(int i,bool limit_high){
if(i==m)return 1;
int an=0,i,j;
int lo=0;
int up=limit_high?s[i]-'0':9;
for(i=lo;i<=up;i++){
an+=dfs(i+1,limit_high&&i==up);
}
return an;
}
//主函数调用
return dfs(0,true);
```

同时有上下界

code:

```
//枚举从 l 到 r
string sl=to_string(l);
string sr=to_string(r);
int m=sr.size();
//将 l 与 r 补齐
while(sl.size()<m){
    sl='0'+sl;
}
int dfs(int i,bool limit_low,bool limit_high){
if(i==m)return 1;
int an=0,i,j;
int lo=limit_low?sl[i]-'0':0;
int up=limit_high?sr[i]-'0':9;
for(i=lo;i<=up;i++){
an+=dfs(i+1,limit_low&&lo==i,limit_high&&up==i);
}
return an;
}
//主函数调用
return dfs(0,true,true);
```

ps: 题目要求对其他数位有其他约束时不能直接对 lo，up 修改，而是在下面 for 循环时改为 max(lo,minlimit),min(hi,maxlimit);

e.g

2：考虑前导 0

只有上界

code:

```
//全局变量
string s=to_string(n);
int size=s.size();
//递归函数
int dfs(int i,int mask,bool is_limit,bool is_num){
if(i==size)return is_num;
if(!is_limit&&is_num&&memo[i][mask]!=-1)return memo[i][mask];
   //is_num 确保前面有数 不会进行跳过当前数位的 dfs
int an=0;
if(!is_num){// 可以跳过当前数位
    an=dfs(i+1,mask,false,false);
}
int j;
int lo=is_num?0:1;//确定下界
int up=is_limit?s[i]-'0':9;//确定上界
for(j=lo;j<=up;j++){
```

```
    if((mask>>j&1)==0){
        an+=dfs(i+1,mask|(1<<j),is_limit&&j==up,true);
    }
}
if(!is_limit&&is_num)
memo[i][mask]=an;
return an;
}
return dfs(0,0,true,false);
```

同时拥有上下界

code:

```
//枚举从 l 到 r
string sl=to_string(l);
string sr=to_string(r);
int m=sr.size();
//将 l 与 r 补齐
while(sl.size()<m){
    sl='0'+sl;
}
int dfs(int i,bool limit_low,bool limit_high, bool is_num){
if(i==m)
{
    return 1;//如果包括 0
    //如果不包括 0 则写成  return is_num;
 }
int an=0,i,j;
if(!is_num&&sl[i]=='0') //前面填的都是 0，limit_low 一定是 true
 an+=dfs(i+1,true,false,false);
int lo=limit_low?sl[i]-'0':0;
int up=limit_high?sr[i]-'0':9;
int down=is_num?0:1;//考虑前导 0 的话前面已经处理过前面数位都是 0 的情况，此时要修改
 ↪ 下界
for(i=max(lo,down);i<=up;i++){
an+=dfs(i+1,limit_low&&lo==i,limit_high&&up==i,true);
}
return an;
}
//主函数调用
return dfs(0,true,true,false);
```

3：例子

code:

```cpp
//全局变量
vector<vector<int>> memo(m, vector<int>(1 << 10, -1)); // -1 表示没有计算过  记忆化
↪ 数组
string s=to_string(n);
int size=s.size();

//递归函数
int dfs(int i,int mask,bool is_limit,bool is_num){
if(i==size)return is_num;
if(!is_limit&&is_num&&memo[i][mask]!=-1)return memo[i][mask];
   //is_num 确保前面有数 不会进行跳过当前数位的 dfs
int an=0;
if(!is_num){// 可以跳过当前数位
    an=dfs(i+1,mask,false,false);
}
int j;
int up=is_limit?s[i]-'0':9;//确定上界
for(j=is_num?0:1;j<=up;j++){
    if((mask>>j&1)==0){
        an+=dfs(i+1,mask|(1<<j),is_limit&&j==up,true);
    }
}
if(!is_limit&&is_num)
memo[i][mask]=an;
return an;
}


//主函数
    return dfs(0,0,true,false);
```

i 代表当前来的位

mask 代表前面选的数的集合（用位运算记录每一位是否选过）

mask 表示前面选过的数字集合，换句话说，第 i 位要选的数字不能在 mask 中。

isLimit 表示当前是否受到了 n 的约束（注意要构造的数字不能超过 n）。若为真，则第 i 位填入的数字至多为 s[i]，否则可以是 9。如果在受到约束的情况下填了 s[i]，那么后续填入的数字仍会受到 n 的约束。例如 n=123，如果 i=0 填的是 1 的话，i=1 的这一位至多填 2。如果 i=0 填的是 1，i=1 填的是 2，那么 i=2 的这一位至多填 3。

isNum 表示 i 前面的数位是否填了数字。若为假，则当前位可以跳过（不填数字），或者要填入的数字至少为 1；若为真，则要填入的数字可以从 0 开始。例如 n=123，在 i=0 时跳过的话，相当于后面要构造的是一个 99 以内的数字了，如果 i=1 不跳过，那么相当于构造一个 10 到 99 的两位数，如果 i=1 跳过，相当于构造的是一个 9 以内的数字。

为什么要定义 isNum？因为 010 和 10 都是 10，如果认为第一个 0 和第三个 0 都是我

们填入的数字，这就不符合题目要求了，但 10 显然是符合题目要求的。

code:

```
string sl, sr;
int m;
ll memo[13][13];
ll re[10] = { 0 };
ll dfs(int i, int sum,int num,bool low_limit, bool high_limit, bool is_num) {
    ll an = 0;
    if (i == m)
    {
        return sum;
    }
    if (!low_limit && !high_limit && is_num && memo[i][sum] != -1)
    {
        return memo[i][sum];
    }
    if (!is_num && sl[i] == '0') an += dfs(i + 1,sum,num ,true, false, false);
    int lo = low_limit ? sl[i] - '0' : 0;
    int hi = high_limit ? sr[i] - '0' : 9;
    int down = is_num ? 0 : 1;
    for (int j = max(down, lo); j <= hi; j++) {
        an += dfs(i + 1, sum + (j == num), num, low_limit && j == lo, high_limit &&
↪   j == hi, true);
    }
    if (!low_limit && !high_limit && is_num) {
        memo[i][sum] = an;
    }
    return an;
}
int main() {
    cin >> sl >> sr;
    m = sr.size();
    while (sl.size() < m)sl = '0' + sl;
    int i;
    for (i = 0; i < 10; i++) {
        memset(memo, -1, sizeof(memo));
        re[i] = dfs(0, 0, i, true, true, false);
    }
    for (int i = 0; i < 10; i++)cout << re[i] << ' ';
```

4 经验

运用场景 给定一个范围内（l,r）范围内满足某某条件的数的个数或者数的总和

如果不统计个数的话 看情况是否用 dfs 某一个参数存 sum(即 memo 多开一维存 sum，一定要把 sum 当做一维可变参数不然会错)

考虑用个数的信息整合出所要的 sum，返回值 cnt 与 sum 的 pa

2.0.8 状压 dp

状压 dp 设计一个整型可变参数 status，利用 status 的位信息，来表示：某个样本是否还能使用，然后利用这个信息进行尝试

写出尝试的递归函数 → 记忆化搜索 → 严格位置依赖的动态规划 → 空间压缩等优化

如果有 k 个样本，那么表示这些样本的状态，数量是 2^k 所以可变参数 status 的范围：0 ~ (2^k) - 1

样本每增加一个，状态的数量是指数级增长的，所以状压 dp 能解决的问题往往样本数量都不大 一般样本数量在 20 个以内 (10^6)，如果超过这个数量，计算量（指令条数）会超过 10^7 ~ 10^8

如果样本数量大到状压 dp 解决不了，或者任何动态规划都不可行，那么双向广搜是一个备选思路

1. 将一堆数分为若干个子集的状压

f(sta,cur,line)

sta 表示位图表示的哪些火柴能选哪些不能选

cur 表示当前这条边还差多少长度选够

line 表示还有多少条边要选

但是 dp 表只要 sta 这一维因为 sta 能够决定 cur 和 line

2. 枚举 status 的所有子集

如 100100 的子集有 100100,100000,000100

方法

```
for(int j=status;j>0;j=(j-1)&status){}
```

ps: 状态中不包含 0，如果需要要额外添加

2.0.9 最长递增子序列

暴力方法 o(n2)

优化 o(nlogn)

1. 严格递增

code：

```
   int findnum(vector<int>end,int l,int r,int num){
    int m,an=-1;
    while(r>=l){
        m=(l+r)>>1;
        if(end[m]>=num){
            an=m;
            r=m-1;
        }
        else l=m+1;
    }
    return an;
}
        int n=nums.size();
         vector<int>dp(m+1);//此时 dp 数组也可以省略 答案为 r-l+1
        vector<int>end(m+1);//end[i] 代表长度为 i 的递增数组最小的最后一个数
        int l=1,r=1;
        end[1]=nums[0];
        int i,find,num;
        for(i=1;i<n;i++){
            num=findnum(end,l,r,nums[i]);
            if(num==-1)end[++r]=nums[i];
            else end[num]=nums[i];
        }
        return r-l+1;
```

2.0.9.1 2. 非递减子序列 只需要修改二分方法，其他一样

code：

```
int findnum(vector<int>end,int l,int r,int num){
    int m,an=-1;
    while(r>=l){
        m=(l+r)>>1;
        if(end[m]>num){
            an=m;
            r=m-1;
        }
        else l=m+1;
    }
    return an;
}
```

2.0.9.2 最少上升子序列覆盖数 一个序列，把它划分成几个“严格递减”的子序列，每个数必须出现在某一个子序列里，并且不能重复。

做法：先 reverse，维护一个 multiset，先插入 a[1]，每次贪心 upper_bound 找到比自己大的最小的数，然后删除（如果没有就不删）

然后把 a[i] 插进去

一个结论

Dilworth 定理 是偏理论的结论，意思是：

> 在一个偏序集（例如：数字按大小比较）中，最少需要多少个下降子序列来覆盖，等于最长上升子序列的长度。

反过来也成立：

- 最少上升子序列数 = 最长下降子序列长度

2.0.9.3 最长递增子序列的个数 首先离散化，然后维护一个 pii 树状数组 tr，tr[i] 表示以 i 为结尾的最长递增子序列长度和个数，从左往右遍历的时候，查询 sign-1 左边的 pii 找到最大的以及累加次数，那么此处的 pair 就是长度 +1，个数

3 图论

3.0.1 2-sat

建好的边就类似一个蕴含等值式

建 2-SAT 图，必须将所有肯定关系都找出建边，否则可能会出纰漏

1. 我们将一个元素拆成两个点表示 bool 元素的两种情况（点 i 和点 n+i），有向边表示若起点成立，则终点一定成立。

2. 当拆点建图后，如果一个元素拆出的两个点 u,v 。存在有向图上的路径（u >v ，则点 u 可以推出点 v ，点 u 非法，则点 v 合法。

3. 有向无环图的情况下，合法点的拓扑序比非法点大。

4. Tarjan 后同一元素拆点强连通分量编号小的点是合法点。

5. 如果一个元素拆成的两个点在同一个强连通分量里，即强连通分量编号相同，那么整个序列无解。

6. 如果一个元素拆成的两个点之间没有任何路径相连，即使是有向路径，那么这两点都可以成为合法点，这两点的分量编号不同，选小的即可。

题目需要的是 a v b，可以转化为 (-a->b)&(-b->a)

```
int dfn[N+1],low[N+1],scc[N+1];
bool instk[N+1];
stack<int>stk;
vector<int>g[N+1];
int n,m,cnt,scc_cnt;
void tarjan(int u){
    dfn[u]=low[u]=++cnt;
    instk[u]=true;
    stk.push(u);
    for(auto v:g[u]){
        if(!dfn[v]){
            tarjan(v);
            low[u]=min(low[u],low[v]);
        }
        else if(instk[v]){
            low[u]=min(low[u],dfn[v]);
        }
    }
    if(low[u]==dfn[u]){
        scc_cnt++;
        while(stk.top()!=u){
            int x=stk.top();
            stk.pop();
```

```
            scc[x]=scc_cnt;
            instk[x]=false;
        }
        stk.pop();
        scc[u]=scc_cnt;
        instk[u]=false;
    }
}
//main
 fill(instk,instk+2*n+1,false);
    for(int i=1;i<=m;i++){
        int a,b,c,d;
        cin>>a>>b>>c>>d;
        g[b?a:a+n].push_back(d?c+n:c);
        g[d?c:c+n].push_back(b?a+n:a);
    }
    for(int i=1;i<=2*n;i++){
        if(!dfn[i])tarjan(i);
    }
    for(int i=1;i<=n;i++){
        if(scc[i]==scc[i+n]){
            cout<<"IMPOSSIBLE";
            return 0;
        }
    }
    cout<<"POSSIBLE"<<'\n';
    for(int i=1;i<=n;i++){
        cout<<(scc[i]<scc[i+n]?0:1)<<' ';
    }
```

3.0.2 Bellman-ford

Bellman-Ford 算法，解决可以有负权边但是不能有负环（保证最短路径存在）的图，单源最短路径算法

松弛操作 假设源点为 A，从 A 到任意点 F 的最短距离为 distance[F] 假设从点 P 出发某条边，去往点 S，边权为 W 如果发现，distance[P] + W < distance[S]，也就是通过该边可以让 distance[S] 变小 那么就说，P 出发的这条边对点 S 进行了松弛操作

Bellman-Ford 过程 1、每一轮考察每条边，每条边都尝试进行松弛操作，那么若干点的 distance 会变小 2、当某一轮发现不再有松弛操作出现时，算法停止

```
vector<int>dis(n+1,inf);
    dis[1]=0;
    bool is_relax=false;
    for(int i=1;i<=n;i++){
        is_relax=false;
        for(auto [x,y,z]:graph){
            if(dis[x]!=inf&&dis[x]+z<dis[y]){
                dis[y]=dis[x]+z;
                is_relax=true;
            }
        }
        if(!is_relax){
            break;
        }
    }
}
```

Bellman-Ford 算法时间复杂度 假设点的数量为 N，边的数量为 M，每一轮时间复杂度 O(M) 最短路存在的情况下，因为 1 次松弛操作会使 1 个点的最短路的边数 +1 而从源点出发到任何点的最短路最多走过全部的 n 个点，所以松弛的轮数必然 ≤ n 1 所以 Bellman-Ford 算法时间复杂度 O(M * N)

重要推广：判断从某个点出发能不能到达负环 上面已经说了，如果从 A 出发存在最短路（没有负环），那么松弛的轮数必然 ≤ n 1 而如果从 A 点出发到达一个负环，那么松弛操作显然会无休止地进行下去 所以，如果发现从 A 点出发，在第 n 轮时松弛操作依然存在，说明从 A 点出发能够到达一个负环

3.0.2.1 +SPFA 优化 Bellman-Ford + SPFA 优化（Shortest Path Faster Algorithm）很容易就能发现，每一轮考察所有的边看看能否做松弛操作是不必要的 因为只有上一次被某条边松弛过的节点，所连接的边，才有可能引起下一次的松弛操作 所以用队列来维护“这一轮哪些节点的 distance 变小了”下一轮只需要对这些点的所有边，考察有没有松弛操作即可

SPFA 只优化了常数时间，在大多数情况下跑得很快，但时间复杂度为 O(n * m) 看复杂度就知道只适用于小图，根据数据量谨慎使用，在没有负权边时要使用 Dijkstra 算法

不加 spfa 不需要建图，加了则需要建图

```
list<pa>graph[2001];
    vector<bool>inqueue(n+1);
    vector<int>times(n+1,0);
    vector<int>dis(n+1,inf);
    fill(inqueue.begin(),inqueue.end(),false);
    queue<int>qu;
    qu.push(1);
    inqueue[1]=true;
    times[1]++;
    dis[1]=0;
    while(!qu.empty()){
        int x=qu.front();
        qu.pop();
        inqueue[x]=false;
        for(auto [y,z]:graph[x]){
            if(dis[x]!=inf&&dis[y]>dis[x]+z){
                dis[y]=dis[x]+z;
                if(!inqueue[y]){
                    inqueue[y]=true;
                    qu.push(y);
                    times[y]++;
                    if(times[y]>n-1){//最多松弛 n-1 轮 如果轮数超过 n-1 就存在负环
                        cout<<"YES"<<endl;
                        return ;
                    }
                }
            }
        }
    }
    cout<<"NO"<<endl;
```

3.0.3 Floyd

Floyd 算法，得到图中任意两点之间的最短距离 时间复杂度 $O(n^3)$，空间复杂度 $O(n^2)$，常数时间小，容易实现 适用于任何图，不管有向无向、不管边权正负，但是不能有负环（保证最短路存在）

过程简述： `distance[i][j]` 表示 i 和 j 之间的最短距离 `distance[i][j] = min(distance[i][j], distance[i][k] + distance[k][j])` 枚举所有的 k 即可

也可以跑 n 次 dj 实现只是要求边权非负，而 floyd 不管正负

code：

```
for(int k=1;k<=n;k++){
        for(int i=1;i<=n;i++){
            for(int j=1;j<=n;j++){
                graph[i][j]=min(graph[i][j],graph[i][k]+graph[k][j]);
            }
        }
    }
//注意循环顺序
```

3.0.4 割点

1. 是根节点并且有 >=2 个儿子
2. 不是根节点且 low[v]>=dfn[u]

```
bool is_p[N + 1];
int dfn[N + 1], low[N + 1];
void tarjan(int u, int f) {
    int child = 0;
    dfn[u] = low[u] = cnt++;
    for (auto v : g[u]) {
        if (!dfn[v]) {
            tarjan(v, u);
            child++;
            low[u] = min(low[u], low[v]);
            if (f != 0 && low[v] >= dfn[u]) {
                is_p[u] = true;//是割点
            }
        }
        else if (v != f) low[u] = min(low[u], dfn[v]);
    }
    if (f == 0 && child >= 2)is_p[u] = true;
}
```

3.0.5 桥

low[v]>dfn[u]

注意有重边，存祖先的边的 id 不走重边

```
bool is_bridge[M+1];
int dfn[N+1],low[N+1];
int n,m,cnt=1;
void tarjan(int u,int f){
    dfn[u]=low[u]=cnt++;
    for(auto [v,id]:g[u]){
        if(!dfn[v]){
            tarjan(v,id);
            low[u]=min(low[u],low[v]);
            if(low[v]>dfn[u])is_bridge[id]=true;
        }
        else if(id!=f){
            low[u]=min(low[u],dfn[v]);
        }
    }
}
 for(int i=1;i<=m;i++){
        int u,v;
        cin>>u>>v;
        g[u].push_back({v,i});
        g[v].push_back({u,i});
    }
    for(int i=1;i<=n;i++){
        if(!dfn[i])tarjan(i,0);
    }
    int an=0;
    for(int i=1;i<=m;i++)an+=is_bridge[i];
    cout<<an;
```

3.0.6 强联通分量

```
list<int>graph[N + 1];
int dfn[N + 1] = { 0 };
int low[N + 1] = { 0 };
bool instack[N + 1];
vector<vector<int>>an;//储存强联通分量
stack<int>st;
int cnt = 1;
void dfs(int u) {
    dfn[u] = low[u] = cnt++;
    instack[u] = true;
    for (auto v : graph[u]) {
        if (dfn[v] == 0) {
            dfs(v);
            low[u] = min(low[u], low[v]);
        }
        else if (instack[v]) {
            low[u] = min(low[u], dfn[v]);
        }
    }
    if (dfn[u] == low[u]) {
        vector<int>temp;
        int x;
        do {
            x = st.top();
            st.pop();
            temp.push_back(x);
            instack[x] = false;
        } while (x != u);
        an.push_back(temp);
    }
}
    int n, m, u, v;
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        cin >> u >> v;
        if (u != v)graph[u].push_back(v);
    }
    fill(instack, instack + n + 1, false);
    for (int i = 1; i <= n; i++) {
        if (dsu[i] == 0)dfs(i);
    }
```

3.0.7 欧拉路径

1. 欧拉路径定义：

图中经过所有边恰好一次的路径叫欧拉路径（也就是一笔画）。如果此路径的起点和终点相同，则称其为一条欧拉回路。

2. 欧拉路径判定（是否存在）：

- 有向图欧拉路径：图中恰好存在 1 个点出度比入度多 1（这个点即为 起点 S），1 个点入度比出度多 1（这个点即为 终点 T），其余节点出度 = 入度。
- 有向图欧拉回路：所有点的入度 = 出度（起点 S 和终点 T 可以为任意点）。
- 无向图欧拉路径：图中恰好存在 2 个点的度数是奇数，其余节点的度数为偶数，这两个度数为奇数的点即为欧拉路径的 起点 S 和 终点 T。
- 无向图欧拉回路：所有点的度数都是偶数（起点 S 和终点 T 可以为任意点）。

判断有向图欧拉路径

```
const int N=1e5;
int n,m;
vector<int>graph[N+1];
int in[N+1];
int out[N+1];
int now[N+1];
int fa[N+1];
stack<int>st;
int find(int x){
    if(fa[x]!=x){
        fa[x]=find(fa[x]);
    }
    return fa[x];
}
void un(int x,int y){
    int fx=find(x);
    int fy=find(y);
    if(fx>fy){
        swap(fx,fy);
        swap(x,y);
    }
    fa[fy]=fx;
}
bool issame(int x,int y){return find(x)==find(y);}
bool connect(){
    for(int i=1;i<=n;i++)fa[i]=i;
    for(int i=1;i<=n;i++){
        for(auto v:graph[i])un(i,v);
    }
    for(int i=1;i<=n;i++){
```

```
        if(!issame(i,1))return false;
    }
    return true;
}
void dfs(int u){
    if(graph[u].size()>0)
    {
        for(int i=graph[u][now[u]];now[u]<graph[u].size();i=graph[u][now[u]]){
            now[u]++;
            dfs(i);
        }
    }
    st.push(u);
}
signed main(){
    ios::sync_with_stdio(false);
    cin.tie(0),cout.tie(0);
    int u,v;
    cin>>n>>m;
    for(int i=1;i<=m;i++){
        cin>>u>>v;
        graph[u].push_back(v);
        out[u]++;
        in[v]++;
    }
    bool ok=connect();
    int sta=-1,en=-1;
    for(int i=1;i<=n;i++){
        if(abs(in[i]-out[i])>=2){
            ok=false;break;
        }
        else if(in[i]-out[i]==1){
            if(en==-1)en=i;
            else{
                ok=false;break;
            }
        }
        else if(out[i]-in[i]==1){
            if(sta==-1)sta=i;
            else{ok=false;break;}
        }
    }
    if(!ok){
        cout<<"No"<<endl;
    }
    else{
        for(int i=1;i<=n;i++)sort(graph[i].begin(),graph[i].end());
```

```
            if(sta!=-1)
            dfs(sta);
            else dfs(1);
        while(!st.empty()){
            int u=st.top();
            st.pop();
            cout<<u<<' ';
        }
    }
}
```

首先要利用并查集判断图的联通性，然后再一遍深搜，最后要倒序输出，欧拉通路从出度-入度大于一的地方开始 dfs，回路的话任选一点（如从一开始即可）；

3.0.8 双联通分量

对于无向图

点双连通分量（Vertex-Biconnected Component，简称 V-BCC）一个极大子图，任意两个点至少有两条点不重复的路径（即删除任意一个点仍然连通）。

边双连通分量（Edge-Biconnected Component，简称 E-BCC）一个极大子图，任意两个点至少有两条边不重复的路径（即删除任意一条边仍然连通）。

一个顶点在只能在一个 ebcc 中，但可以在多个 vbcc 中

3.0.8.1 边双联通分量 方法 1：

首先求出所有的桥，然后再 dfs 一遍不往桥上走得到的联通分量就是 ebcc

```cpp
const int N=5e5,M=2e6;
vector<pii>g[N+1];
vector<int>ebcc[N+1];
int low[N+1],dfn[N+1];
bool is_bridge[M+1],vis_ebcc[N+1];
int n,m,cnt=1,ebcc_cnt=0;
void tarjan(int u,int f){
    dfn[u]=low[u]=cnt++;
    for(auto[v,id]:g[u]){
        if(!dfn[v]){
            tarjan(v,id);
            low[u]=min(low[u],low[v]);
            if(low[v]>dfn[u])is_bridge[id]=true;
        }
        else if(f!=id){
            low[u]=min(low[u],dfn[v]);
        }
    }
}
void dfs(int u){
    vis_ebcc[u]=true;
    ebcc[ebcc_cnt].push_back(u);
    for(auto[v,id]:g[u]){
        if(vis_ebcc[v]||is_bridge[id])continue;
        dfs(v);
    }
}

//main
 for(int i=1;i<=m;i++){
        int u,v;
        cin>>u>>v;
        g[u].push_back({v,i});
        g[v].push_back({u,i});
    }
    fill(is_bridge,is_bridge+m+1,false);
    fill(vis_ebcc,vis_ebcc+n+1,false);
    for(int i=1;i<=n;i++){
        if(!dfn[i])tarjan(i,0);
    }
    for(int i=1;i<=n;i++){
        if(!vis_ebcc[i]){
            ebcc_cnt++;
            dfs(i);
        }
    }
    cout<<ebcc_cnt<<'\n';
```

```
    for(int i=1;i<=ebcc_cnt;i++){
        cout<<ebcc[i].size()<<' ';
        for(auto x:ebcc[i])cout<<x<<' ';
        cout<<'\n';
    }
```

方法 2：

把无向边看作有向边，过程就和求强联通分量一样

```
stack<int>stk;
void tarjan(int u,int f){
    dfn[u]=low[u]=cnt++;
    stk.push(u);
    for(auto[v,id]:g[u]){
        if(!dfn[v]){
            tarjan(v,id);
            low[u]=min(low[u],low[v]);
        }
        else if(f!=id){
            low[u]=min(low[u],dfn[v]);
        }
    }
    if(dfn[u]==low[u]){
        ebcc_cnt++;
        while(stk.top()!=u){
            int x=stk.top();
            stk.pop();
            ebcc[ebcc_cnt].push_back(x);
        }
        ebcc[ebcc_cnt].push_back(u);
        stk.pop();
    }
}
```

3.0.8.2 点双联通分量 先给出两个性质：

1. 两个点双最多只有一个公共点，且一定是割点。
2. 对于一个点双，它在 DFS 搜索树中 dfn 值最小的点一定是割点或者树根。

我们根据第二个性质，分类讨论：

1. 当这个点为割点时，它一定是点双连通分量的根，因为一旦包含它的父节点，他仍然是割点。
2. 当这个点为树根时：
    1. 有两个及以上子树，它是一个割点。

2. 只有一个子树，它是一个点双连通分量的根。
3. 它没有子树，视作一个点双

一个点可能会在多个 vbcc 中

```
vector<int>vbcc[N+1];
int dfn[N+1],low[N+1];
int n,m,cnt,vbcc_cnt;
stack<int>stk;
void tarjan(int u,int f){
    dfn[u]=low[u]=++cnt;
    stk.push(u);
    if(f==0&&g[u].empty()){
        vbcc_cnt++;
        vbcc[vbcc_cnt].push_back(u);
        return;
    }
    for(auto [v,id]:g[u]){
        if(!dfn[v]){
            tarjan(v,id);
            low[u]=min(low[u],low[v]);
            if(low[v]>=dfn[u]){
                vbcc_cnt++;
                while(stk.top()!=v){//注意弹栈是弹到 v 而不是 u 因为 u 点 dfs v
                ↪ 之前可能还 dfs 过一些点，那些点可能和 u 在同一个 vbcc 里，如
                ↪ 果弹到 u 就会把这些点也弹出了
                    int x=stk.top();
                    stk.pop();
                    vbcc[vbcc_cnt].push_back(x);
                }
                vbcc[vbcc_cnt].push_back(v);
                stk.pop();
                vbcc[vbcc_cnt].push_back(u);
            }
        }
        else if(id!=f)low[u]=min(low[u],dfn[v]);
    }
}
```

3.0.9 链式前向星

```
//有 n 个点 m 条边
int head[n+1];//head[i] 指的是 i 为出发点的第一条边是第几条边 初始化全为 0 后根据插
↪ 进来的边来更改
//head[i] 和 to[i] 都是第 i 条边的信息
int next[m+1];//如果 next[i] 为 0 终止 储存的是同一开头下一条边的位置
int to[m+1];//指的是该条边指向的位置
//如果边有权值要加 value 数组
int value[m+1];//指的是该条边的权值
int cnt=1;//记录读进来边的数量 每读一条边 cnt++
//初始化
void build(int n){
    fill(head+1,head+n+1,0);
}
//加边
void addedge(int u,int v,int w){
    next[cnt]=head[u];
    to[cnt]=v;
    value[cnt]=w;
    head[u]=cnt++;
}
//便利所有以 p 开头的边
for(int p=head[temp];p>0;p=next[p]){
    int u=temp;//开头
    int v=to[p];//终点
    int w=value[p];//权值
}
```

3.1 树

3.1.1 树的直径

定义：树上距离最远的两个点，形成的路径叫做树的直径

此时的距离可以带权

树的直径的两种求法

1. 两次 dfs(适用于边权非负的树)

从树的根节点走到离根节点最远的点 x，再从 x 走到离 x 最远的点 y，那么 x->y 就是一条直径

```
int n;
int fa[N+1],dis[N+1];//fa 数组用来记录路径
void dfs(int u,int f,int len){
    fa[u]=f;
    dis[u]=dis[f]+len;
    for(auto[v,w]:g[u]){
        if(v!=f)dfs(v,u,w);
    }
}

//main
  }
    dfs(1,0,0);
    int c=1,maxlen=0;
    for(int i=1;i<=n;i++){
        if(dis[i]>maxlen){
            maxlen=dis[i];
            c=i;
        }
    }
    dfs(c,0,0);
    int an=0;
    for(int i=1;i<=n;i++)an=max(an,dis[i]);
    cout<<an<<endl;
```

2. 树形 dp

```
int dis[N+1],down[N+1];//dis[u] 表示必须经过 u 节点的最大路径 down[u] 表示从 u 向下
↪ 的最大路径
void dfs(int u,int f){
    for(auto&[v,w]:g[u]){
        if(v!=f){
            dfs(v,u);
            dis[u]=max(dis[u],down[u]+down[v]+w);
            down[u]=max(down[u],w+down[v]);
        }
    }
}
//main
dfs(1,0);
    int an=0;
    for(int i=1;i<=n;i++)an=max(an,dis[i]);
    cout<<an<<endl;
```

比较

两次 dfs 可以得到路径但是处理不了负权图，树形 dp 可以处理负权图但是不能得到路径

一些如果边权都为为正的结论

如果有多条直径，那么这些直径一定有共同的中间部分，可能是一个公共点或一段公共路径

树上任意一点，相隔最远的点的集合，直径的两个端点至少有一个在这个集合里面

3.1.2 树的重心

以某个点为根时，1. 最大子树的节点数最少 2. 每棵子树的节点数不超过总结点数的一半 3. 所有节点都走向重心的总边数最少

那么这个是重心

3.1.2.1 重心的性质 一棵树最多有两个重心，如果有两个则他们相邻，并且一个重心的最大子树的子树的头结点是另外一个重心

如果树增加或者删除一个叶节点，那么重心最多移动一条边

如果把两棵树连在一起，那么新树的重心一定原来两棵树重心的路径上

树上的边权如果都为正，不管边权的大小，所有节点走向重心的路径和最小

```
vector<int>g[N+1];
vector<int>centroid;
int siz[N+1],maxsub[N+1];
int n;
void get_centroid(int u,int f){
    for(auto v:g[u]){
        if(v!=f){
            get_centroid(v,u);
            maxsub[u]=max(maxsub[u],siz[v]);
            siz[u]+=siz[v];
        }
    }
    maxsub[u]=max(maxsub[u],n-siz[u]);
    if(maxsub[u]<=n/2)centroid.push_back(u);
}


//main
    fill(siz+1,siz+n+1,1);
    get_centroid(1,0);
    sort(all(centroid));
```

```
    for(auto x:centroid)cout<<x<<' ';
```

3.1.3 树上倍增与 LCA

倍增算法建表 O(nlogn) 每次查询 logn，tarjanO(n)

但是 tarjan 是离线的，你把所有问题给完之后遍历一遍树得到所有答案

```
const int N = 5e5;
vector<int>g[N + 1];
int dep[N + 1];
int fa[N + 1][32];
int maxdep = 0;
void dfs(int u, int f) {
    fa[u][0] = f;
    if (f != -1)dep[u] = dep[f] + 1;
    maxdep = max(maxdep, dep[u]);
    for (int i = 1; i < 32; i++) {
        if (fa[u][i - 1] == -1)fa[u][i] = -1;
        else fa[u][i] = fa[fa[u][i - 1]][i - 1];
    }
    for (auto v : g[u]) {
        if (v != f)dfs(v, u);
    }
}
int same_dep(int u, int diff) {
    for (int i = 0; i <= (int)log2(diff); i++) {
        if ((diff >> i) & 1)u = fa[u][i];
    }
    return u;
}
int lca(int a, int b) {
    if (dep[a] < dep[b])swap(a, b);
    a = same_dep(a, dep[a] - dep[b]);
    if (a == b) return a;
    for (int i = maxdep; i >= 0; i--) {
        if (fa[a][i] != fa[b][i]) {
            a = fa[a][i], b = fa[b][i];
        }
    }
    return fa[a][0];
```

```
}

//main
dep[s] = 1;
    fa[s][0] = -1;
    dfs(s, -1);
    maxdep = (int)log2(maxdep);
    while (q--) {
        int a, b;
        cin >> a >> b;
        cout << lca(a, b) << '\n';
    }
```

3.1.3.1 使用倍增算法求 LCA

3.1.3.2 使用 tarjan 算法求 LCA 算法过程： 1）处理所有问题，建好每个节点的问题列表，然后遍历树 2）来到当前节点 cur，令 visited[cur] = true，表示当前节点已经访问 3）遍历 cur 的所有子树，每棵子树遍历完都和 cur 节点合并成一个集合，集合设置 cur 做头节点 4）遍历完所有子树后，处理关于 cur 节点的每一条查询 (cur, x) 如果发现 x 已经访问过，cur 和 x 的最低公共祖先 = x 所在集合的头节点 如果发现 x 没有访问过，那么当前查询先不处理，等到 x 节点时再去处理查询 (x, cur) 得到答案

```
int an[N+1],fa[N+1];
bool vis[N+1];
vector<int>g[N + 1];
vector<pii>qs[N+1];
int find(int x){
    if(x!=fa[x]){
        fa[x]=find(fa[x]);
    }
    return fa[x];
}
void un(int x,int y){
    x=find(x),y=find(y);
    fa[y]=x;
}
void tarjan(int u){
    vis[u]=true;
    for(auto v:g[u]){
        if(!vis[v]){
            tarjan(v);
            un(u,v);
        }
    }
    for(auto[x,y]:qs[u]){
```

```
        if(vis[x]){
            an[y]=find(x);
        }
    }
}

//main
for(int i=0;i<q;i++){
        int x,y;
        cin>>x>>y;
        qs[x].push_back({y,i});
        qs[y].push_back({x,i});
    }
    fill(vis,vis+n+1,false);
    iota(fa,fa+n+1,0);
    tarjan(s);
    for(int i=0;i<q;i++)cout<<an[i]<<'\n';
```

一个结论

如果边权为正，假设 lca(a,b)=c，那么 a 到 b 路径长度 = 头结点到 a 的长度 + 头结点到 b 的长度-头结点到 lca(a,b) 的长度 *2

3.1.4 树上差分

3.1.4.1 树上点差分 修改两点之间路径上点的所有权值

方法：

修改点权：

num[x]+=v,num[y]+=v,num[lca(x,y)]-=v,num[fa[lca(x,y)]]-=v

最后只用 dfs 一遍，每个根节点加上所有子树的权值和

3.1.4.2 树上边差分 修改两点之间路径上边的所有权值

方法：

还是修改点权：

num[x]+=v,num[y]+=v,num[lca(x,y)]-=2*v

最后 dfs 一遍，每条边加上子节点的点权

边的权值等于边指向儿子节点的权值

3.1.5 最小斯坦纳树

问题：求 k 个节点的最小生成树，但是有一些额外节点可以选择

总时间复杂度是 O(n×3^k+mlogm×2^k)

流程

目标：在一般带权无向连通图中，找到一个连通子图把给定端点集合 S 全部连起来，且边权和最小。允许使用非端点作为中转（“Steiner 点”）。

由于 $k \le 10$，使用经典的 Dreyfus-Wagner 子集 DP。

令

$$\mathrm{dp}[M][v]$$

表示：连接端点子集 $M \subseteq S$ 的最小代价，且该连通子图“收敛于 / 终止在”结点 v。

直觉：把所有需要连起来的端点按子集 M 分组，最终的树可以看作若干子树在某个点 v 合并。

转移（两步）

1）子集合并（只在同一终止点 v 合并）

$$\mathrm{dp}[M][v] \leftarrow \min_{\emptyset \neq M_1 \subset M} \Big(\mathrm{dp}[M_1][v] + \mathrm{dp}[M \setminus M_1][v] \Big)$$

含义：两棵覆盖不相交端点集的最优子树在 v 合并。

2）图上松弛（把“终止点”从 u 走到 v，类似于一个换根过程）

$$\mathrm{dp}[M][v] \leftarrow \min_{u} \Big(\mathrm{dp}[M][u] + \mathrm{dist}[u][v] \Big)$$

其中 dist 是图上任意两点的最短路长度（如用 Dijkstra 预处理或每步多源 Dijkstra 传播）。
这一步等价于允许我们用图的最短路径把“合并点”移动到更合适的位置。

初始（单端点 s_i）

$$\mathrm{dp}[1 \ll i][v] = \mathrm{dist}[s_i][v]$$

即覆盖集合只含端点 s_i 时，让“终止点”在任意 v，代价就是从 s_i 到 v 的最短路长度（把“终止点在 s_i”的状态一次性松到全图）。

答案

$$\min_{v} \ \mathrm{dp}[\mathrm{FullMask}][v], \qquad \mathrm{FullMask} = (1 \ll k) - 1$$

复杂度

常见实现的时间复杂度：

$$O\big(n \cdot 3^k + m \cdot 2^k \log n\big).$$

板子：

```
const int N=100;
vector<pii>g[N+1];
vector<int>nums;
int dp[1<<10][N];
priority_queue<pii,vector<pii>,greater<pii>>pq;
void djstra(int s){
    while(!pq.empty()){
        auto[d,u]=pq.top();
        pq.pop();
        if(d!=dp[s][u])continue;
        for(auto &[v,w]:g[u]){
            if(w+d<dp[s][v]){
                dp[s][v]=w+d;
                pq.push({dp[s][v],v});
            }
        }
    }
}
    int n,m,k;
    cin>>n>>m>>k;
    for(int i=1;i<=m;i++){
        int u,v,w;
        cin>>u>>v>>w;
        u--,v--;
        g[u].push_back({v,w});
        g[v].push_back({u,w});
    }
    nums.resize(k);
    for(int i=0;i<k;i++){
        cin>>nums[i];
        nums[i]--;
    }
    sort(all(nums));
    memset(dp,0x3f,sizeof(dp));
    for(int i=0;i<k;i++)dp[1<<i][nums[i]]=0;
    for(int s=0;s<(1<<k);s++){
        for(int i=0;i<n;i++){
            for(int j=s;j>0;j=s&(j-1)){
                dp[s][i]=min(dp[s][i],dp[j][i]+dp[s^j][i]);
            }
        }
        for(int i=0;i<n;i++){
```

```
            if(dp[s][i]!=inf)pq.push({dp[s][i],i});
        }
        djstra(s);
    }
    int an=inf;
    for(int i=0;i<n;i++)an=min(an,dp[(1<<k)-1][i]);
    cout<<an;
```

4 数学

4.0.1 矩阵快速幂

给定一个 k * k 的矩阵乘 n 次 时间复杂度是 O(logn * k^3)

```
vector<vector<int>> multiply(vector<vector<int>>&a,vector<vector<int>>&b){
    int n=a.size(),m=b[0].size(),p=b.size();
    vector<vector<int>>an(n,vector<int>(m));
    for(int i=0;i<n;i++){
        for(int j=0;j<m;j++){
            for(int k=0;k<p;k++){
                an[i][j]+=a[i][k]*b[k][j];
            }
        }
    }
    return an;
}
vector<vector<int>> ksm(vector<vector<int>>&a,int b){
    int n=a.size();
    vector<vector<int>>re(n,vector<int>(n,0));
    for(int i=0;i<n;i++)re[i][i]=1;
    while(b){
        if(b&1)re=multiply(re,a);
        b>>=1;
        a=multiply(a,a);
    }
    return re;
}
```

ps ： 矩阵快速幂只能计算正方形矩阵

4.0.1.1 1. 固定关系的 1 维 k 阶递推表达式 即 dp 表只有一维参数，dpi=x1dpi-1+……. xkdpi-k

其中 xi 可以为任何数包括 0

方法：

首先给出一个 1 行 k 列的矩阵表示初始值 {dpk,……. dp1}，然后乘以一个 k*k 的矩阵 n-k 次，得到矩阵 {dpn……dpn-k+1}

k*k 矩阵的第一列为 {x1…. xk}

剩下的系数通过带入前几项得到

e. g. 斐波那契数 f0=0, f1=1

初始矩阵 {1，0}，关系矩阵 {{1，1}，{x,y}}

待定系数求出 x=1，y=0

然后求 n-1 次关系矩阵然后初始矩阵再乘以关系矩阵

```
if(n==0)return 0;
        vector<vector<int>>re;
        vector<int>te={1,0};
        re.push_back(te);
        vector<vector<int>>b={{1,1},{1,0}};
        b=ksm(b,n-1);
        re=multiply(re,b);
        return re[0][0];
```

4.0.1.2 2. 固定关系的 k 维 1 阶递推表达式 其中 dp 表有两维参数,dpi,j=x1dpi-1,0+……xkdpi-1,k-1（j>=0&&j<k)

方法：

和上几乎一样

k 维则取一个 k*k 矩阵，然后第 p 行 q 列系数则为 dpi,q-1 表达式中 xp 来决定

此时整个矩阵可以直接得到不需要待定系数了

4.0.2 逆元

4.0.2.1 连续数字逆元的线性递推 inv[i] 表示 i 的逆元

```
inv[1]=1;
inv[i]=mod-1ll*inv[mod%i]*(mod/i)%mod;//从左向右线性递推
```

```
po[0]=1;
for(int i=1;i<=n;i++)po[i]=1ll*po[i-1]*i%mod;
inv[i]=ksm(po[i],mod-2);
for(int i=n-1;i>=0;i--)inv[i]=1ll*(i+1)*inv[i+1]%mod;
```

4.0.2.2 连续阶乘逆元的线性递推

4.0.3 判断较大数字是否是质数（Miller-Rabin 测试）

```cpp
// 判断较大的数字是否是质数 (Miller-Rabin 测试)
// C++ 同学可以提交如下代码
// 可以通过所有测试用例，核心在于第 11 行，整型成了 128 位
// 测试链接 : https://www.luogu.com.cn/problem/U148828

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

typedef __int128 ll;
typedef pair<int, int> pii;

template<typename T> inline T read() {
    T x = 0, f = 1; char ch = 0;
    for(; !isdigit(ch); ch = getchar()) if(ch == '-') f = -1;
    for(; isdigit(ch); ch = getchar()) x = (x << 3) + (x << 1) + (ch - '0');
    return x * f;
}

template<typename T> inline void write(T x) {
    if(x < 0) putchar('-'), x = -x;
    if(x > 9) write(x / 10);
    putchar(x % 10 + '0');
}

template<typename T> inline void print(T x, char ed = '\n') {
    write(x), putchar(ed);
}

ll t, n;

ll qpow(ll a, ll b, ll mod) {
    ll ret = 1;
    while(b) {
        if(b & 1) ret = (ret * a) % mod;
        a = (a * a) % mod;
        b >>= 1;
    }
    return ret % mod;
}

vector<ll> p = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};

bool miller_rabin(ll n) {
    if(n < 3 || n % 2 == 0) return n == 2;
```

```
    ll u = n - 1, t = 0;
    while(u % 2 == 0) u /= 2, ++ t;
    for(auto a : p) {
        if(n == a) return 1;
        if(n % a == 0) return 0;
        ll v = qpow(a, u, n);
        if(v == 1) continue;
        ll s = 1;
        for(; s <= t; ++ s) {
            if(v == n - 1) break;
            v = v * v % n;
        }
        if(s > t) return 0;
    }
    return 1;
}

int main() {
    t = read<ll>();
    while(t --) {
        n = read<ll>();
        if(miller_rabin(n)) puts("Yes");
        else puts("No");
    }
    return 0;
}
```

4.0.4 质因数分解

```
int n;
vector<int>prime;
for(int i=2;i*i<=n;i++){
if(n%i==0){
prime.push_back(i);
}
while(n%i==0)n/=i;
}
if(n!=1)prime.push_back(n);
```

ps：注意最后一行

4.0.5 质数筛

```
int n;
bool isprime[n+1]={true};
vector<int>prime;
for(int i=2;i<=n;i++){
if(isprime[i]){
prime.push_back(i);
for(int j=i*i;j<=n;j+=i)isprime[j]=false;
}
}
```

4.0.5.1 埃式筛（时间复杂度 O(nloglogn)

```
bool isprime[N + 1];//判断一个数是不是质数 true 代表是
vector<int>prime;//所有的质数集合
int miny[N + 1];//所有数的最小质因子
miny[1] = 1;
for (i = 1; i <= N; i++)isprime[i] = true;
for (i = 2; i <= N; i++) {
    if (isprime[i]) {
        prime.push_back(i);
        miny[i] = i;
    }
    for (j = 0; j < prime.size() && i * prime[j] <= N; j++) {
        isprime[i * prime[j]] = false;
        miny[i * prime[j]] = prime[j];//最小的质因子就是被欧筛的那个素数
        if (i % prime[j] == 0)break;
    }
}
```

```
//可用该代码求 n 的所有质因子
 for (int mod = miny[n]; mod != 1; mod = miny[n /= mod])
```

4.0.5.2 欧拉筛（时间复杂度 O(n)）

4.0.6 中位数

1. 问题模型

数轴 x 上有 n 个点，现在给出这 n 个点的坐标 ai，让你选择一个点 k(k∈[1,n])，使得每个点到点 k 的距离之和最小。

2. 结论

点 k 为序列的中位数时最优：若 n 为奇数，点 k 位于 a[(n+1)/2] 处最优；若 n 为偶数，点 k 位于 a[n/2] 或 a[n/2+1] 均为

3. 推导

首先将 a[1]~a[n] 排序，假设点 p 选在坐标 q 处，点 p 左侧有 l 个点，点 p 右侧有 r 个点，此时分为两种情况：l<r，若点 p 向右移动单位距离，则距离之和减小 (r l)。r<l，若点 p 向左移动单位距离，则距离之和减小 (l r)。

因此保证点 p 的左侧节点和右侧节点数量尽量相等时（即 l=r），距离之和最优。

该结论同样适用于求出最优点 k，当点 k 的左右两侧节点数相等时为最优解，此时 k 就是序列的中位数。此时无论左移或右移点 k 都会使结果更差。

4.0.7 带权中位数

1. 问题模型

数轴 x 上有 n 个点，现在给出这 n 个点的坐标 ai以及这 n 个点的点权 num[i]，让你选择一个点 k(k∈[1,n])，使得每个点到点 k 的距离与点权的乘积之和最小。

2. 结论

满足 Σki=1num[i]≥tot/2 时（tot 为所有点权之和）的最小点 k 为最优点。

3. 推导

可以把每个节点 i 由原来只有一个节点看做有 num[i] 个点同时位于坐标 i 上，此时我们知道当点 k 位于 tot/2 的坐标上最优。

4.0.8 组合数和快速幂

```
const int mod=998244353;
const int N=4e6;
int  po[N+1];
int  inv[N+1];
int  ksm(int a,int b){
    int re=1;
    while(b){
        if(b&1)
        re=1ll*re*a%mod;
        a=1ll*a*a%mod;
        b>>=1;
    }
    return re;
}
int  C(int n,int m){
    if(n<0||m<0||n<m)return 0;
    if(n==m)return 1;
    return 1ll*po[n]*inv[m]%mod*inv[n-m]%mod;
}




    po[0]=1;
    for(int i=1;i<=N;i++){
        po[i]=1ll*po[i-1]*i%mod;
    }
    inv[N]=ksm(po[N],mod-2);
    for(int i=N-1;i>=0;i--)inv[i]=1ll*inv[i+1]*(i+1)%mod;
```

4.0.9 博弈

1. 公平组合游戏（ICG）

ICG:

1，两个玩家轮流且游戏方式一致

2，两人的选择都是最优

3，游戏一定会在有限局内结束（以玩家无法行动结束）

1.bash game

一共 n 颗石子，两人轮流拿，每次可以拿 1~m 颗，拿到最后一颗的获胜

结论：n!=(m+1) 的倍数先手赢 反之后手

如果 n==(m+1) 倍数 先手无论拿多少后手可以再将石头堆拿至 (m+1) 倍数

2.nim game

一共 n 堆石头，两人轮流进行

玩家需要选择一个非空石头堆，从该石头堆移除任意整数石头

谁先拿走最后石头就赢

结论：若所有石头数量异或和不等于 0 先手赢 反之后手

若只有两堆石头 有相同数量，后手总可以模仿先手做镜像动作则后手必赢

3.SG 函数和 SG 定理

SG 函数求解过程：

最终必败点是 A，规定 SG（A）=0 假设状态点是 B，那么 SG（B）= 查看 B 所有后继节点的 sg 值，其中没有出现过的最小自然数 SG（B）! = 0，那么状态 B 为必胜态；SG（B）== 0，那么状态 B 为必败态

SG 定理（Bouton 定理）

如果一个 ICG 游戏（总），由若干个独立的 ICG 子游戏构成（分 1、分 2、分 3..），那么：SG（总）= SG（分 1）^ SG（分 2）^SG（分 3）任何 ICG 游戏都是如此，正确性证明类似尼姆博弈

当数据规模较大时，要善于通过对数器的手段，打印 SG 表并观察，看看能不能发现简洁规律

5 字符串

5.0.1 ac 自动机

ac 自动机功能

给你若干目标字符串，还有一篇文章，返回每个目标字符串在文章中出现了几次

fail 指针含义：

此时这个节点一定是某个目标字符串的前缀，设置这个前缀为 s，要在所有目标串的前缀中找一个是 s 的最长真后缀，将当前节点的 fail 指针指向最长真后缀的尾部节点

经典 ac 自动机

首先将目标字符串建立 trie，然后利用 bfs 层序遍历建立 fail 指针，0 节点的 fail 指向自己，当遍历到某个节点的时候，设置它儿子的 fail 指针：假设当前节点指向儿子的路为 a，依次沿着 fail 指针向上移动直到有一个节点指向儿子的路也为 a 的，将之前那个儿子指向这个儿子，如果一直找不到就指向 0

遍历文章：首先从 0 开始，到文章的 x 字符，如果当前节点有指向 x 的路，那么就移动到 x，并把 x 的 fail 指针沿途所有节点的 times++，如果没有指向 x 的路，那么就沿着 fail 向上找哪个节点有指向 x 的路，如果找到了就移动到 x，并把 x 的 fail 指针沿途所有节点的 times++，如果一直找不到的话那么就节点设置为 0

最后每个单词出现过的词频即为这个单词最后一个字符的 times

优化 ac 自动机

优化 fail 指针建立过程

将原始建立的 trie 改变，如果当前节点 tr[i] [x] 为 0，那么将 tr[i] [x] 设置为 tr[fail[i]] [x]，避免了绕圈

优化收集词频的时候

添加词频的时候只添加当前节点的词频，然后把当前节点与 fail 建立反图，最后类似树形 dp 从底到顶收集

优化提前跳出

如果是给出一些敏感词然后如果遇到一个敏感词就直接跳出的话，那么优化二就没必要了，我们此时设置一个 alert 数组，alert[i]=true 表示这个点是一个单词的结尾，此时我们设置 alert[i]|=alert[fail[i]]（因为 fail 节点的字符串是 son 字符串的后缀，并且原本一个点要加词频它的所有 fail 都加），那么只需要遍历句子的时候当前节点的的 alert 为 true 就跳出

用了优化 1 和优化 2

```
const int N=2e5;
vector<int>g[N+1];
int tr[N+1][26];
int fail[N+1];
int times[N+1];
int in[N+1];
int cnt=0;
int n;
int add(string s){
    int cur=0;
    for(auto c:s){
        int num=c-'a';
        if(tr[cur][num]==0)tr[cur][num]=++cnt;
        cur=tr[cur][num];
    }
    return cur;
}
void get_fail(){
    queue<int>qu;
    for(int i=0;i<26;i++){
        if(tr[0][i]!=0)qu.push(tr[0][i]);
    }
    while(!qu.empty()){
        int m=qu.size();
        for(int i=0;i<m;i++){
            int cur=qu.front();
            qu.pop();
            for(int j=0;j<26;j++){
                if(tr[cur][j]==0){
                    tr[cur][j]=tr[fail[cur]][j];
                }
                else{
                    int cur2=tr[cur][j];
                    qu.push(cur2);
                    fail[cur2]=tr[fail[cur]][j];
                }
            }
        }
    }
}
void add_times(string s){
    int cur=0;
    for(auto c:s){
        int num=c-'a';
        cur=tr[cur][num];
        times[cur]++;
    }
```

```
    for(int i=1;i<=cnt;i++){g[i].push_back(fail[i]);in[fail[i]]++;}
    queue<int>qu;
    for(int i=1;i<=cnt;i++){
        if(in[i]==0)qu.push(i);
    }
    while(!qu.empty()){
        int u=qu.front();
        qu.pop();
        for(auto v:g[u]){
            if(--in[v]==0)qu.push(v);
            times[v]+=times[u];
        }
    }
}
    vector<int>en;
    for(int i=1;i<=n;i++){
        cin>>s;
        en.push_back(add(s));
    }
    get_fail();
    cin>>s;
    add_times(s);
    for(auto x:en)cout<<times[x]<<'\n';
```

用了优化 1 和优化 3

```
int tr[N+1][10];
int fail[N+1];
bool alert[N+1];
int cnt=0;
void add(string s){
    int cur=0;
    for(int i=0;i<s.size();i++){
        int num=s[i]-'0';
        if(tr[cur][num]==0)tr[cur][num]=++cnt;
        cur=tr[cur][num];
    }
    alert[cur]=true;
}
void get_fail(){
    queue<int>qu;
    for(int i=0;i<10;i++){
        if(tr[0][i]!=0){
            qu.push(tr[0][i]);
        }
    }
    while(!qu.empty()){
```

```
        int siz=qu.size();
        for(int i=0;i<siz;i++){
            int cur=qu.front();
            qu.pop();
            for(int j=0;j<10;j++){
                int cur2=tr[cur][j];
                if(cur2==0){
                    tr[cur][j]=tr[fail[cur]][j];
                }
                else{
                    qu.push(cur2);
                    fail[cur2]=tr[fail[cur]][j];
                    alert[cur2]|=alert[fail[cur2]];// 其他和之前都差不多多加了这一句
                }
            }
        }
    }
}
```

5.0.2 kmp

匹配大字符串 s1 中是否含有 s2

next 数组（对 s2 求）

含义：不包含当前，前面的字符串前缀和后缀（不能包含整体，例如只有一个字符的时候不能算）最大匹配长度

但是可以交错，e.g aaaaat 在 t 位置的 next 值为 4

利用 next 数组加速匹配

如果此时 s1 的开头在 i,s2 位置在 cur，匹配 s1,s2 这个字符不相等了，那么就把 s2 的开头放到 i+cur-next[cur] 的位置，以及此时从 s2 的 next[cur] 位置开始匹配

新开头省了匹配过程

因为前缀等于后缀，那么相等的那一段就不用匹配了，直接从 s2 的 next[cur] 位置开始匹配

淘汰不可能的出发点

为什么 i 到 i+cur-next[cur] 之间不用作为开始，假如这两点之间存在一个点作为开头能完整的匹配出 s2，则 next 数组一定可以被这个点作为后缀开头更新的更大

板子

```
vector<int>get_next(string s2){
    int m=s2.size();
    vector<int>next(m);
    next[0]=-1;
    if(m==1)return next;//注意这里
    next[1]=0;
    int i=2,j=0;
    while(i<m){
        if(s2[i-1]==s2[j])next[i++]=++j;
        else if(j>0)j=next[j];
        else next[i++]=j;
    }
    return next;
}
int kmp(string s1,string s2){
    int n=s1.size(),m=s2.size();
    vector<int>next=get_next(s2);
    int i=0,j=0;
    while(i<n&&j<m){
        if(s1[i]==s2[j]){i++;j++;}
        else if(j>0)j=next[j];
        else i++;
    }
    return j==m?i-j:-1;
}
```

小拓展用法

next 数组求的是最长公共前后缀，那么去递归求 next[next[i]]，next[next[next[i]]]……就可以得到所有的公共前后缀长度

num[i] 表示的是以 0~i-1 为最长前缀获得的总的公共前后缀个数 注意这里的定义和 next 数组有点不一样，这里 0_{i-1} 已经是最长前缀了，而 next 数组是 0^{i-1} 区间的最长前缀

初始化 num[1]=1 解释：如果一个字符串是以 s[0] 作为前缀的话，自己本身已经是一个公共前后缀了，（这里要注意上面 num 数组的定义）

```
void get_next(){
    nxt[0]=-1;
    nxt[1]=0;
    num[1]=1;
    int i=2,j=0;
    while(i<=n){
        if(s[i-1]==s[j]){
            j++;
            nxt[i]=j;
            num[i]=num[j]+1;
            i++;
        }
        else if(j>0)j=nxt[j];
        else{
            nxt[i]=j;
            num[i]=num[j]+1;
            i++;
        }
    }
}
```

5.0.3 拓展 kmp

z 数组定义：

z[i] 表示从 i 位置开始与原字符串的最长前缀，显然的 z[0]= 整个字符串长度

和 manacher 几乎同理

流程：

当来到出发点 i

1.i 没有被 r 包住，那么以 i 为出发点直接扩展

2.i 被 r 包住，前点 i-c 的扩展范围在大拓展范围之内，那么 z[i]=z[i-c]

3.i 被 r 包住，前点 i-c 的扩展范围在大拓展范围之外，那么 z[i]=r-i

4.i 被 r 包住，前点 i-c 的扩展范围刚好在边界上，那么从 r 位置接着往右扩展

```
const int N=1e5;
int z[N+1];
void get_zarray(string s){
    int n=s.size();
    z[0]=n;
    for(int i=1,c=1,r=1,len;i<n;i++){
        len=r>c?min(z[i-c],r-i):0;
        while(i+len<n&&s[i+len]==s[len])len++;
        if(i+len>r){
            r=i+len;
            c=i;
        }
        z[i]=len;
    }
}
```

e 数组定义：

有两个字符串 a,b

e[i] 表示 a 从 i 出发，b 从 0 出发的最长公共前缀

流程：

先求出 b 字符串的 z 数组

其他的把对称点

```
const int N=1e5;
int z[N+1],e[N+1];
void get_earray(string a,string b){
    get_zarray(b);
    int n=a.size(),m=b.size();
    for(int i=0,r=0,c=0,len;i<n;i++){
        len=r>i?min(r-i,z[i-c]):0;
        while(i+len<n&&len<m&&a[i+len]==b[len])len++;
        if(i+len>r){
            r=i+len;
            c=i;
        }
        e[i]=len;
    }
}
```

5.0.4 manacher

求解字符串的最长回文子串，时间复杂度 O(n)

前置：

首先要获得拓展字符串，在 back 和 front 位置以及两个字符中间加字符 #，整体字符串长度将变为 2*n+1(这一步可以方便偶回文串的计算，不需要虚轴的概念）

回文半径数组 p：从某个点开始（包含那个点）向左右两边扩获得回文串，此时的半径长度。回文半径-1= 最长回文子串长度

回文覆盖最右边界 r：前面回文字符串的最右边界 +1

回文中心 c：回文覆盖最右边界的所在字符串的中心，如果 r 相同 c 取最左的

拓展回文串的最右端下标/2= 原字符串最右端下标/2+1

流程：

1：如果 r==i，那么此时得不到加速，以 i 为中心暴力拓展

2. 如果 r>i，对称点 2c-i 的回文半径在大半径里面，那么 p[i]=p[2c-i]

3. 如果 r>i，对称点 2c-i 的回文半径在大半径外面，那么 p[i]=r-i

4. 如果 r>i，对称点 2c-i 的回文半径刚好与大半径重合，那么从 r 位开始接着扩展

板子：

```
int manacher(string&s){
    int n=s.size();
    string ss(2*n+1,'#');
    for(int i=1,j=0;j<n;j++,i+=2)ss[i]=s[j];
    n=2*n+1;
    vector<int>p(n);
    int an=0;
    for(int r=0,c=0,len,i=0;i<n;i++){
        len=r>i?min(r-i,p[2*c-i]):1;
        while(i+len<n&&i-len>=0&&ss[i+len]==ss[i-len])len++;
        if(i+len>r){
            r=i+len;
            c=i;
        }
        p[i]=len;
        an=max(an,len);
    }
    return an-1;
}
```

如果要回推原字符串的最右端，先求出最长回文长度 len=p[pos]-1;pos+=len 得到加长串的最右端，然后 len=len/2-1 得到的就是原字符串的最右端

5.0.5 trie

实现前缀树 Trie 类：

1. Trie() 初始化前缀树对象。
2. void insert(String word) 将字符串 word 插入前缀树中。
3. int search(String word) 返回前缀树中字符串 word 的实例个数。
4. int prefixNumber(String prefix) 返回前缀树中以 prefix 为前缀的字符串个数。
5. void delete(String word) 从前缀树中移除字符串 word。

code

```
const int N=1e5;
int tr[N][26];
int p[N];
int e[N];
int cnt=1;
void insert(string s){
    int n=s.size();
    int cur=1;
    p[cur]++;
    for(int i=0;i<n;i++){
        int num=s[i]-'a';
        if(tr[cur][num]==0){
            tr[cur][num]=++cnt;
        }
        cur=tr[cur][num];
        p[cur]++;
    }
    e[cur]++;
}
int search(string s){
    int n=s.size();
    int cur=1;
    for(int i=0;i<n;i++){
        int num=s[i]-'a';
        if(tr[cur][num]==0)return 0;
        cur=tr[cur][num];
    }
    return e[cur];
}
int pre(string s){
    int n=s.size();
    int cur=1;
    for(int i=0;i<n;i++){
        int num=s[i]-'a';
        if(tr[cur][num]==0)return 0;
        cur=tr[cur][num];
```

```
    }
    return p[cur];
}
void cut(string s){
    if(search(s)>0){
        int cur=1;
        int m=s.size();
        for(int i=0;i<m;i++){
            int num=s[i]-'a';
            if(--p[tr[cur][num]]==0){
                tr[cur][num]=0;
                return;
            }
            cur=tr[cur][num];
        }
        e[cur]--;
    }
}
//ps cut 函数的剪枝要防止有路线走到剪枝的后面，以防万一还是写成 p[tr[cur][num]]--;
```

ps：遇到数字类 如果数字特别大，tr 第二维没必要开特别大，可以将一个数拆开，如“9987“可以拆成”9987#“，用‘#’表示一个数字已经结束了，则 tr 第二维只需要开[10];

5.0.6 字符串哈希

字符串哈希：如何得到整个字符串的哈希值

1）理解 long 类型自然溢出，计算加、减、乘时，自然溢出后的状态等同于对 2 的 64 次方取模的值状态 2）字符串转化成 base 进制的数字并让其自然溢出 3）base 可以选择一些质数比如：433、499、599、1000000007 也可以选择已经被验证实了很好用的值：31、131、1313、13131、131313 等

建议选择质数，不要选经典值，因为会被出题人刻意构造碰撞

4）转化时让每一位的值从 1 开始，不从 0 开始，这样就得到了一个 long 类型的数字代表字符串 5）利用数字的比较去替代字符串的比较，可以大大减少复杂度

字符串哈希理论上说会有碰撞导致出错，但现实中的算法考察样本量太少了，出错概率非常低。即便是出错了，也可以更换进制数 base，再去赌，一定能赌赢。没错！是玄学！但是好用！堪称赌狗的胜利

预处理

```
const int p=31/131/....
ull po[N+1],po1[N+1]

int n=s.size();

string s=" "+s;

po[0]=1

for(int i=1;i<=n;i++)po[i]=po[i-1]*p;
for(int i=1;i<=n;i++)po1[i]=po1[i-1]*p+s[i]-'a'+1;
//计算一个区间 l 到 r 的值
po1[r]-po1[l-1]*po[r-l+1]
```

6 分块

6.0.1 块状数组

原理

假设数据规模为 n，块的大小和块的数量都是根号 n 规模

分块维护信息时，散块信息暴力维护，整块信息维护 lazy 就可以

进行区间修改和区间查询的时候，只要操作左散块和右散块和中间整块

适用场景

区间不可合并的信息，线段树等结构难以维护，分块往往容易

建块的过程

```
int blen,bnum,n;
int bi[N+1],bl[M+1],br[M+1];
void build(){
    blen=sqrt(n);
    bnum=(n+blen-1)/blen;
    for(int i=1;i<=n;i++)bi[i]=(i+blen-1)/blen;
    for(int i=1;i<=bnum;i++){
        bl[i]=(i-1)*blen+1;
        br[i]=min(i*blen,n);
    }
}
```

板子题

a 为原数组，b 为在块里排序的数组，对于查询：如果不是整块就暴力查询，复杂度根号 n，如果是整块就在排序后的数组二分找

对于修改，先在 a 中修改后再把 a 赋值给 b 让 b 重新排序

```
void add(int x,int ad){
    a[x]=ad;
    int l=bl[bi[x]],r=br[bi[x]];
    for(int i=l;i<=r;i++)b[i]=a[i];
    sort(b+l,b+r+1);
}
int query(int l,int r,int x){
    int an=0;
    if(bi[l]==bi[r]){
        for(int i=l;i<=r;i++){
            if(a[i]>=x)an++;
        }
    }
    else{
        for(int i=l;i<=br[bi[l]];i++){
            if(a[i]>=x)an++;
        }
        for(int i=bl[bi[r]];i<=r;i++){
            if(a[i]>=x)an++;
        }
        for(int i=bi[l]+1;i<=bi[r]-1;i++){
            int l=bl[i],r=br[i];
            an+=find(l,r,x)-l+1;
        }
    }
    return an;
}
    for(int i=1;i<=bnum;i++){
        sort(b+bl[i],b+br[i]+1);
    }
```

6.0.2 树上分块

6.0.2.1 重链序列分块

将整棵树重链剖分，然后基于 dfn 序将序列进行分块，每个块用一个 bitset 来存，一条链上重链数量为 $logn$，每次查询一条重链为 $\sqrt{n}$，分为左散 + 中间 + 右散，单次查询是 $\sqrt{n}logn$ 的

```
const int N=4e4,B=1e3;
int a[N+1];
vector<int>g[N+1];
int n,q;
int blen,bnum,bi[N+1],bl[B+1],br[B+1];
bitset<N>bt[B+1];
int dfn[N+1],seg[N+1],fa[N+1],son[N+1],top[N+1],siz[N+1],dep[N+1],dfn_cnt;
vector<int>nums;
void build(){
    blen=sqrt(30*n);
    bnum=(n+blen-1)/blen;
    for(int i=1;i<=n;i++){
        bi[i]=1+(i-1)/blen;
    }
    for(int i=1;i<=bnum;i++){
        bl[i]=1+(i-1)*blen;
        br[i]=min(n,i*blen);
    }
    for(int i=1;i<=bnum;i++){
        for(int j=bl[i];j<=br[i];j++)
        bt[i][a[seg[j]]]=1;
    }
}
bitset<N>query(int l,int r){
    bitset<N>an;
    if(bi[l]==bi[r]){
        for(int i=l;i<=r;i++)an[a[seg[i]]]=1;
    }
    else{
        for(int i=l;i<=br[bi[l]];i++)an[a[seg[i]]]=1;
        for(int i=bl[bi[r]];i<=r;i++)an[a[seg[i]]]=1;
        for(int i=bi[l]+1;i<=bi[r]-1;i++)an|=bt[i];
    }
    return an;
}
signed main()
{
    ios::sync_with_stdio(false);
    cin.tie(0), cout.tie(0);
    cin>>n>>q;
    for(int i=1;i<=n;i++){
        int x;
        cin>>x;
        a[i]=x;
        nums.push_back(x);
    }
    sort(all(nums));
```

```
    nums.erase(unique(all(nums)),nums.end());
    for(int i=1;i<=n;i++){
        a[i]=lower_bound(all(nums),a[i])-nums.begin();
    }
    for(int i=1;i<=n-1;i++){
        int u,v;
        cin>>u>>v;
        g[u].push_back(v);
        g[v].push_back(u);
    }
    dfs1(1,0);//dfs1 和 dfs2 同重链剖分
    dfs2(1,1);
    build();
    int an=0;
    while(q--){
        int x,y;
        cin>>x>>y;
        x^=an;
        if(x>y)swap(x,y);
        bitset<N>te;
        while(top[x]!=top[y]){
            if(dep[top[x]]>dep[top[y]])swap(x,y);
            te|=query(dfn[top[y]],dfn[y]);
            y=fa[top[y]];
        }
        te|=query(min(dfn[x],dfn[y]),max(dfn[x],dfn[y]));
        an=te.count();
        cout<<an<<'\n';
    }
    return 0;
```

6.0.2.2 树上随机撒点 树上随机选 $\sqrt{n}$ 个关键点，非关键点到它最近的关键点，两个相邻的关键点，期望距离都是 $\sqrt{n}$

非关键点到它最近的关键点之间视为散块，关键点到关键点之间视为整块，类似于之前的整块散块结合

关键点需要维护：

1. 它是第几个关键点 2. 向上跳到的关键点，3. 下到上的位图

marknum 表示关键点的数量，vis[i] 表示第 i 号节点是否已经是关键点了，用于随机撒的过程，nodek[k]=i 表示第 k 个关键点是 i,node[i]=k 表示第 i 号节点是第 k 个关键点，如果 k 为 0 表示不是，up[i]=j 表示第 i 号节点是关键点，它向上跳的最近的关键点是第 j 号节点，bt[k] 表示第 k 个关键点及其上面点（没到下一个关键点）组成的位图

类似于分块，对于整块就利用储存的位图，对于散块就去暴力计算，还需要倍增求 lca

```
const int N=1e5,M=3e4,P=17,B=5e3;
int a[N+1],st[N+1][P+1],dep[N+1],nodek[B+1],node[N+1],up[N+1];
bitset<M>bt[B+1];
bool vis[N+1];
vector<int>g[N+1];
int n,m,c,maxdep,blen,bnum;
bitset<M>te;
//dfs 和 lca 同树上倍增
void build(){
    dfs(1,0);
    maxdep=(int)log2(maxdep);
    blen=sqrt(n*10);
    bnum=(n+blen-1)/blen;
    fill(vis,vis+n+1,false);
    for(int i=1;i<=bnum;i++){
        int x;
        do{
            x=rd(1,n);
        }while(vis[x]);
        nodek[i]=x;
        node[x]=i;
        vis[x]=true;
    }
    for(int i=1;i<=bnum;i++){
        int x=nodek[i];
        bt[i][a[x]]=1;
        x=st[x][0];
        while(x){
            if(node[x]){
                up[nodek[i]]=x;
                break;
            }
            bt[i][a[x]]=1;
            x=st[x][0];
        }
    }
}
void query(int x,int y){
    while(node[x]==0&&x!=y){
        te[a[x]]=1;
        x=st[x][0];
    }
    while(up[x]>0&&dep[up[x]]>dep[y]){
        te|=bt[node[x]];
        x=up[x];
```

```
    }
    while(x!=y){
        te[a[x]]=1;
        x=st[x][0];
    }
    te[a[y]]=1;
}
```

6.0.3 块状链表

适用于区间增删，区间移动的题目

实现上，不用动态链表实现，利用栈 pool 进行块编号分配和回收，本质上是个静态数组

规定块的容量为 2* 根号 n，每个块里字符的数量 <= 容量

插入操作时：先分裂成左右两块，在两块之间插入若干新块

删除操作时：分裂最左块和最右块，删除最左和最右和中间

进行完插入和删除操作后，遍历检查相邻块大小，如果内容和小于容量就合并

```
const int N = 4e6, blen = 4e3, bnum = 2e3;
struct node {
    int siz, nxt = -1;
    string s;
}b[bnum + 1];
vector<int>use;
pii find(int x) {
    if (x == 0)return { 0,0 };
    int sum = 0, cur = b[0].nxt, cur2 = 0;
    while (sum + b[cur].siz < x) {
        sum += b[cur].siz;
        cur = b[cur].nxt;
    }
    cur2 = x - sum;
    return { cur,cur2 };
}
void split(int bi, int bi2) {
    if (bi == 0 || bi2 == 0)return;
    int te = use.back();
    use.pop_back();
    b[te].nxt = b[bi].nxt;
```

```
    b[bi].nxt = te;
    b[te].siz = b[bi].siz - bi2;
    b[bi].siz = bi2;
    b[te].s = b[bi].s.substr(bi2);
    b[bi].s = b[bi].s.substr(0, bi2);
}
void insert(int bi, string s) {
    int n = s.size(), cur = 0;
    while (n) {
        int te = use.back();
        use.pop_back();
        b[te].siz = min(n, blen);
        b[te].s = s.substr(cur, b[te].siz);
        cur += b[te].siz;
        n -= b[te].siz;
        b[te].nxt = b[bi].nxt;
        b[bi].nxt = te;
        bi = te;
    }
}
void merge(int cur) {
    while (cur != -1) {
        while (b[cur].nxt != -1 && b[cur].siz + b[b[cur].nxt].siz <= blen) {
            int te = b[cur].nxt;
            b[cur].s += b[te].s;
            b[cur].siz += b[te].siz;
            b[cur].nxt = b[te].nxt;
            use.push_back(te);
        }
        cur = b[cur].nxt;
    }
}
void erase(int bi, int n) {
    int te = b[bi].nxt;
    int sum = 0;
    while (sum + b[te].siz < n) {
        sum += b[te].siz;
        use.push_back(te);
        te = b[te].nxt;
    }
    split(te, n - sum);
    use.push_back(te);
    te = b[te].nxt;
    b[bi].nxt = te;
}
void print(int bi, int bi2, int n) {
    int te = b[bi].siz - bi2;
```

```
    if (n <= te) {
        cout << b[bi].s.substr(bi2, n);
    }
    else {
        cout << b[bi].s.substr(bi2);
        n -= te;
        bi = b[bi].nxt;
        while (n) {
            int num = min(n, b[bi].siz);
            cout << b[bi].s.substr(0, num);
            n -= num;
            bi = b[bi].nxt;
        }
    }
}
```

7 杂类

7.0.1 分治

- 该问题的规模缩小到一定的程度就可以容易地解决。
- 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。

找到平面内两点间欧几里得距离最小值

```
#include<bits/stdc++.h>
#define pa pair<ll,ll>
using namespace std;
typedef long long ll;
typedef unsigned long long ull;
const int inf =0x3f3f3f3f;
const int N=4e5;
ll an=2e18;
int n;
pa a[N+1];
bool cmp1(pa a,pa b){return a.first<b.first;}
bool cmp2(pa a,pa b){return a.second<b.second;}
ll dis(pa a,pa b){
    return 1ll*(a.first-b.first)*(a.first-b.first)+1ll*(a.second-
    ↪ b.second)*(a.second-b.second);
}
void merge(int l,int r){
    if(l==r)return;
    if(r==l+1){an=min(an,dis(a[l],a[r]));return;}
    int m=(l+r)>>1;
    merge(l,m);
    merge(m+1,r);
    vector<pa>te;
    int sign=m;
    while(sign<=r&&powl(a[sign].first-a[m].first,2)<=an){
        te.push_back(a[sign]);
        sign++;
    }
    sign=m-1;
    while(sign>=l&&powl(a[m].first-a[sign].first,2)<=an){
        te.push_back(a[sign]);
        sign--;
    }
    sort(te.begin(),te.end(),cmp2);
    for(int i=0;i<(int)te.size()-1;i++){
        int sign=i+1;
        while(sign<te.size()&&powl(te[sign].second-te[i].second,2)<=an){
```

```
            an=min(an,dis(te[i],te[sign]));
            sign++;
        }
    }
}
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0), cout.tie(0);
    cin>>n;
    for(int i=1;i<=n;i++)cin>>a[i].first>>a[i].second;
    sort(a+1,a+n+1,cmp1);
    merge(1,n);
    cout<<an<<endl;
    return 0;
}
```

7.0.2 倍增

线段上有 n 个点，给定每个点往右跳一步的最远距离 st[i][0]

已知从任意点出发都能到达最终点，并且 i<j，有 st[i][0]<st[j][0]

如何构建出一张表

st[i][j] 的含义：从 i 号点跳 2^j 次能到达的最远点

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

如何快速计算出两点之间至少跳几步到达

计算到达小于且尽量接近终点的步数然后 +1

```
//假设从 a 到 b
int an=0;
for(int i=31;i>=0;i--){
    if(st[a][i]<b){
        an+=1<<i;
        a=st[a][i];
    }
}
return an+1;
```

7.0.3 对拍器

```
mt19937_64 rg(random_device{}());//种子
int rd(int l,int r){return l+rg()%(r-l+1);}//生成一个 [l,r] 范围为的随机数
ll rdl(ll l,ll r){return l+rg()%(r-l+1);}//ll
```

生成一个序列

```
int n=rd(1,1000);
vector<int>p(n);
for(int i=0;i<n;i++)p[i]=i+1;
shuffle(p.begin(),p.end(),rg);
```

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

using i64 = long long;

std::mt19937 rnd(std::chrono::steady_clock().now().time_since_epoch().count());

int rng(int l, int r) { // [l, r]
    return rnd() % (r - l + 1) + l;
}

void array(int n) { // Generates an array with n elements
    int m = 1E9;
    std::vector<int> a(n);
    for (int i = 0; i < n; i++) {
        a[i] = rng(0, m);
        std::cout << a[i] << " \n"[i == n - 1];
```

```
    }
}

void cand(int m, int n) { // Generate m cands from 1 to n
    for (int i = 0; i < m; i++) {
        int l = rng(1, n);
        int r = rng(1, n);

        if (l > r) {
            std::swap(l, r);
        }

        std::cout << l << " " << r << "\n";
    }
}

void tree(int n) { // Generate a tree with n vertices
    int m = 1E9;
    for (int i = 1; i < n; i++) {
        int p = rng(0, i - 1);
        int v = rng(1, m);
        std::cout << p + 1 << " " << i + 1 << " " << v << "\n";
    }
}

void graph(int n, int m) {
// Generate an undirected graph with n vertices and m edges. There are no double
 ↪  edges or self-rings in the graph, and must be connected.
    std::vector<std::pair<int, int>> e;
    std::map<std::pair<int, int>, bool> f;

    std::cout << n << " " << m << "\n";

    for (int i = 1; i < n; i++) {
        int p = rng(0, i - 1);
        e.push_back(std::make_pair(p, i));
        f[std::make_pair(p, i)] = f[std::make_pair(i, p)] = true;
    }

    // debug(f);
    for (int i = n; i <= m; i++) {
        int x, y;
        do {
            x = rng(0, n - 1);
            y = rng(0, n - 1);
        } while (x == y || f.count(std::make_pair(x, y)));
```

```
        e.push_back(std::make_pair(x, y));
        f[std::make_pair(x, y)] = f[std::make_pair(y, x)] = true;
    }

    std::shuffle(e.begin(), e.end(), rnd);

    for (auto [x, y] : e) {
        std::cout << x + 1 << " " << y + 1 << "\n";
    }
}

int main() {
    int t = 5;
    std::cout << t << "\n";

    while (t--) {
        graph(3, 3);
    }

    return 0;
}
```

7.0.4 pbds

```
#include <bits/stdc++.h>        // C++ STL 万能头
#include <bits/extc++.h>        // PBDS 万能头
using namespace std;
using namespace __gnu_pbds;

// 🔹 有序集合（支持 order_of_key / find_by_order）
template<typename T>
using ordered_set = tree<
    T,
    null_type,
    less<T>,                // 可改为 greater<T> 实现降序
    rb_tree_tag,
    tree_order_statistics_node_update
>;

// 🔹 有序映射（带键值对 + 有序操作）
```

```
template<typename K, typename V>
using ordered_map = tree<
    K,
    V,
    less<K>,
    rb_tree_tag,
    tree_order_statistics_node_update
>;

// ⍰ 快速哈希表（unordered_map 替代，更快防卡）
template<typename K, typename V>
using hash_map = gp_hash_table<K, V>;

// ⍰ 抗卡哈自定义哈希（用于 hash_map）
struct custom_hash {
    static uint64_t splitmix64(uint64_t x) {
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }
    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM =
         ↪  chrono::steady_clock::now().time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }
};
```

ordered_set

```
ordered_set<int> s;
s.insert(5);
s.insert(10);
cout << *s.find_by_order(1) << "\n"; // 输出 10
cout << s.order_of_key(7) << "\n";   // 输出 1（5 < 7）
```

ordered_map

```
ordered_map<int, string> mp;
mp[3] = "apple";
mp[1] = "banana";
auto it = mp.find_by_order(0); // 第 0 小 key
cout << it->first << " = " << it->second << "\n"; // 1 = banana
```

hash_map 示例（默认哈希）

```
hash_map<int, int> cnt;
cnt[1000000000] = 123;
cout << cnt[1000000000] << "\n";
```

hash_map 示例（抗卡哈版本）

```
gp_hash_table<long long, int, custom_hash> mp;
mp[123456789123] = 1;
```

s.find_by_order(k)——找第 k 小的元素（从 0 开始）

函数原型

cpp

```
iterator find_by_order(int k);
```

功能

返回有序集合中排名为第 **k** 小的元素的迭代器。

- 返回值是迭代器，所以用 *s.find_by_order(k) 取值。
- **k** 从 0 开始。

示例

```
ordered_set<int> s = {10, 20, 30, 40};

cout << *s.find_by_order(0) << "\n";  // 10
cout << *s.find_by_order(2) << "\n";  // 30
cout << *s.find_by_order(3) << "\n";  // 40
```

注意事项

- 如果 k >= s.size()，则返回 s.end()（非法访问要小心）。
- 元素是自动升序排序的（less）

s.order_of_key(x) ——找小于 x 的元素个数（也就是 x 的排名）

函数原型

cpp

```
int order_of_key(const T& x);
```

功能

返回集合中严格小于 **x** 的元素个数。

即 x 的排名是第几个（从 0 开始）。

示例

```
ordered_set<int> s = {10, 20, 30, 40};

cout << s.order_of_key(10) << "\n";  // 0 （<10的没有）
cout << s.order_of_key(25) << "\n";  // 2 （10, 20 < 25）
cout << s.order_of_key(40) << "\n";  // 3 （10,20,30 < 40）
```

区间个数查询（重点）

求 [L, R] 闭区间内元素个数：

cpp

```
s.order_of_key(R + 1) - s.order_of_key(L)
```

为什么这样写？

- order_of_key(R + 1)：返回小于等于 R 的个数（因为是 < R+1）
- order_of_key(L)：返回小于 L 的个数
- 相减就是在 [L, R] 中的个数

示例

```
ordered_set<int> s = {5, 10, 15, 20, 25, 30};

// 查询 [10, 25] 中有几个元素
int ans = s.order_of_key(26) - s.order_of_key(10);  // = 5 - 1 = 4
cout << ans << "\n";  // 输出 4，对应：10,15,20,25
```

7.0.5 快读快写

```cpp
// 快读
inline int read() {
    int x = 0, f = 1;
    char c = getchar();
    while (c < '0' || c > '9') { if (c == '-') f = -1; c = getchar(); }
    while (c >= '0' && c <= '9') { x = x * 10 + (c ^ 48); c = getchar(); }
    return x * f;
}

// 快写
inline void write(int x) {
    if (x < 0) putchar('-'), x = -x;
    if (x > 9) write(x / 10);
    putchar(x % 10 + '0');
}
//回车 putchar('\n');
//空格 putchar(' ');
```