```cpp
#include <bits/stdc++.h>
using namespace std;
#define int long long
#define rep(i ,a ,b) for(int i =a;i<=b;++i)
#define endl '\n'
#define debug(a) cout<<#a<<'='<<a<<endl;
#define inf 0x3f3f3f3f
#define ls u<<1
#define rs u<<1|1
typedef pair<int,int> pii;
void solve(){

}


signed main(){
    ios::sync_with_stdio(false);
    cout.tie(0);
    cin.tie(0);
    int _ = 1;
    //cin>>_;
    while(_--)solve();
    return 0;
}
```

基础算法

冒泡排序

```cpp
作者：王清楚
链接：https://ac.nowcoder.com/discuss/1453293
来源：牛客网

for (int i = 0; i < a.size(); ++i)
    {
        for (int j = 0; j + 1 < a.size(); ++j)
        {
            if (a[j].first == a[j + 1].first)continue;
            if (k > 0 && a[j].second < a[j + 1].second)
            {
                swap(a[j], a[j + 1]);
                --k;
            }
        }
    }
```

冒泡排序每交换一个数字会减少一个逆序数

快速排序

令指针i， j 指向数列的区间外侧，数列的中值即为x， 数列中<=x的数放左段， >=x的数放右。 然后对于左右两段，再递归以上过程。

退出while时， i ==j 或i =j+1
相同元素可能会交换，是不稳定的

1. 如果每次选的x能让左右两段近似等分，会生成一颗有lognn层的**均衡二叉树**， 每层的i j 两个指针会便利n个元素 Onlogn
2. 如果每次选的x只能分离出一个元素，会退化成有n层链， On^2

为了避免2这个情况发生，我们可以使用随机化x的方法

```
void quick_sort(int q[] , int l ,int r){
    if(l >= r)return;
    int i =l-1 , j = r+1, x = q[l+r >> 1];
    while(i < j){
        do i ++ ; while (q[i] < x);
        do j-- ; while(q[j] > x );
        if(i < j)swap(q[i] , q[j]);
    }
    quick_sort(q , l ,j) , quick_sort(q ,j+1 , r);
}
```

归并排序

```
void merge_sort(int l ,int r){
    if(l >= r)return;
    int mid = l + r >>1;
    merge_sort(l , mid);
    merge_sort(mid + 1 , r);
    int k =0 , i = l , j = mid+1;
    while(i <= mid && j <= r){
        if(a[i] <= a[j])tmp[k++] = q[i++];
        else tmp[k++] = a[j++];
    }
    while(i <= mid)tmp[k++] = a[i++];
    while(j <= r)tmp[k++] = a[j++];

    for(i = l , j =0;i<=r;++i , ++j)a[i] = tmp[j];

}
```

反悔贪心

```
#include <bits/stdc++.h>
using namespace std;
#define int long long
#define rep(i ,a ,b) for(int i =a;i<=b;++i)
#define endl '\n'
#define debug(a) cout<<#a<<'='<<a<<endl;
#define inf 0x3f3f3f3f
#define ls u<<1
#define rs u<<1|1
typedef pair<int,int> pii;
```

```
const int N = 1e5+9;
struct node{
    int val , t;
    bool operator<(const node & pre){
        if(t != pre.t)return t < pre.t;
        return val > pre.val;
    }

}a[N];
int n;
void solve(){
    cin>>n;
    rep(i ,1 ,n)cin>>a[i].t>>a[i].val;
    sort(a+1 , a+1+n);
    priority_queue<int , vector<int> , greater<int>>q;
    int ans = 0;
    rep(i ,1 ,n){
        if(a[i].t > q.size()){
            ans += a[i].val;
            q.push(a[i].val);
        }
        else{
            ans += a[i].val;
            q.push(a[i].val);
            ans -=q.top();
            q.pop();
        }
    }
    cout<<ans<<endl;

}


signed main(){
    ios::sync_with_stdio(false);
    cout.tie(0);
    cin.tie(0);
    int _ = 1;
    //cin>>_;
    while(_--)solve();
    return 0;
}
```

搜索

组合

```
void dfs(int cnt , int num){
    if(cnt == k+1){
        for(int i =1;i<=k;++i){
            cout<<c[i]<<' ';
        }
        cout<<endl;
```

```
            return;
        }
        for(int i =num;i<=n-1;++i){
            if(!vis[i]){
                vis[i] = 1;
                c[cnt] = i;
                dfs(cnt+1 , i);
                vis[i] = 0;
            }
        }
}
```

二维前缀和 二维差分

二维差分

```
void init(int x1 , int y1 , int x2 , int y2){
    for(int i =x1;i<=x2;++i){
        for(int j =y1;j<=y2;++j){
            pre[i][j]++;
        }
    }
}
```

```
 dif[dx][dy]++ , dif[tx+1][ty+1]++;
 dif[dx][ty+1]-- , dif[tx+1][dy]--;
//dx dy 是左上角的数字  tx ty是右下角

pre[i][j] += pre[i-1][j] + pre[i][j-1] - pre[i-1][j-1] + dif[i][]
```

dp

背包dp

01背包

我们可以设 `dp[i][j]` 表示对于前i个物品， 此时背包容量为j， 背包所装物品的最大价值为 dp[i][j];

```
void solve(){
    cin>>n>>W;
    rep(i , 1 , n)cin>>w[i]>>val[i];
    rep(i , 1 , n){
        rep(j , 0 , W){
            if(j < w[i])f[i][j] = f[i-1][j];
            else{
                f[i][j] = max(f[i-1][j] , f[i-1][j -w[i]] + val[i]);
            }
        }
    }
    cout<<f[n][W]<<endl;
}
```

滚动数组优化

```
for(int i =1;i<=n;++i){
    for(int j =V; j>=v[i];--j){
        f[j] = max(f[j] , f[j - v[i]] + w[i]);
    }
}
这里 vi 表示重量 ， wi 表示价值
```

完全背包

```
rep(i, 1, n){
    for(int j =v[i];j<=V;++j){
        f[j] = max(f[j] , f[j-v[i]] + w[i]);
    }
}
```

多重背包

> 有 N 种物品和一个容量是 V 的背包。第 i 种物品**最多有** si 件，每件体积是 vi，价值是 wi。求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总和最大。

多重背包可以看成01背包和完全背包的组合。

转换为01背包

我们可以考虑将**多个同种物品合成一件物品。**比如，我们有10件t恤，一件占空间2，每件价值20元，我们将**8件t恤**合在一起，就变成了一件1占空间为16的价值160元的t恤。如此一来，多重背包问题就被我们转换为01背包问题。

小优化，转化为01+完全背包

完全背包问题是没见物品全部是无限件。总的背包体积就是V，是已经确定了的，如果我们有一种物品占用体积为V，共有s间，但是s * v >= V,不就代表着我们连 s 件物品**都不可能拿完**背包就已经塞不下了吗?所以这种情况**我们可以转换成完全背包来做，只需要加一个判断就可以了。**

我们看上面转化成01背包是需要枚举一下拿多少件的，而转化为完全背包是不需要枚举多少间的，可以拿我们就拿，所以在时间上会有一些优化。

```
rep(i, 1, n){//v表示体积
    if(s[i] * v[i] > = V){ //转化为完全背包
```

```
        for(int j = v[i];j<=V;++j)f[j] = max(f[j] , f[j-v[i]] + w[i]);
    }
    else{ //转化为01背包
        for(int j =V;j>=v[i];--j){
            for(int k =s[i];k>=0;--k){
                if(j >= k*v[i]){
                    f[j] = max(f[j-k*v[i]] + k*w[i] , f[j]); //s是物品的个数
                }
            }
        }

    }
}
```

那么接下来考虑优化， 可以使用二进制优化 或 单调队列优化

二进制优化

```
cin>>n>>V;
int cnt =0; //记录新物体数
for(int i = 1,a,b,s;i<=n;++i){
    cin>>a>>b>>s; //体积 ， 价值， 数量
    int k =1;
    while(k <= s){
        v[++cnt] = k*a;
        w[cnt] = k*b;
        s -=k;
        k *= 2;
    }
    if(s){
        v[++cnt] = s*a;
        w[cnt] = s * b;
    }
}

for(int i =1;i<=cnt;++i){//01背包
    for(int j =V;j>=v[i];--j){
        f[j]=max(f[j],f[j-v[i]]+w[i]);
    }
}
cout<<f[V];
```

例题： 这题有 完全背包， 分组背包， 01背包， 把它们全部用二进制优化转化成01背包

```
void solve(){
    cin>>t1>>cc>>t2>>ts1>>cc>>ts2;
    int T = 60*(ts1-t1)+ts2-t2;

    cin>>n;
    int cnt = 0;
    rep(i ,1 ,n){
        int t , c , p;
        cin>>t>>c>>p;
        if(p == 0){ //完全背包
            p = inf;
```

```
        }
        int s =1;
        while(s <=p){
            v[++cnt] = s*t;
            w[cnt] = s * c;
            p -=s;
            s <<= 1;
        }
        if(p){
            v[++cnt] = p * t;
            w[cnt] = p * c;
        }

    }

    for(int i =1;i<=cnt;++i){
        for(int j =T;j>=v[i];--j){
            f[j] = max(f[j] , f[j -v[i]] +w[i]);
        }
    }
    cout<<f[T]<<endl;
}
```

LIS

`f[i]` 表示以i结尾的最长上升子序列是多长

On^2

```
rep(i ,1 ,n)f[i] = 1;
int ans = 0;
rep(i ,1 , n){
    rep(j ,1 , i-1){
        if(a[i] > a[j])f[i] = max(f[i] , f[j]+1);
    }
    ans = max(ans , f[i]);
}
```

Onlogn

狄尔斯沃定理： 最少的不上升子序列个数 = 最长上升子序列

https://www.luogu.com.cn/record/185200669

```
void solve(){
    int x;
    int n = 0;
    while(cin>>x){
        a[++n] = x;
    }
    //先求出最长不上升子序列
    int len1 =1;
    f1[1] = a[1];
```

```
    rep(i , 2 , n){
        if(a[i] <= f1[len1])f1[++len1] = a[i];
        else{ //此时 a[i] > f1[len1] -> 找到 a[i] < f1[]的位置
            int idx1 = upper_bound(f1+1 , f1+1+len1 , a[i],greater<int>()) - f1;
            f1[idx1] = a[i];
        }
    }
    //最少的非递增子序列数量 = 最长上升子序列
    int len2 =1;
    f2[1] = a[1];
    rep(i ,2 ,n){
        if(a[i] > f2[len2])f2[++len2] = a[i];
        else{ //此时a[i] <= f2[len2] ->  a[i] >= f2[]
            int idx2 = lower_bound(f2+1 , f2+1+len2 , a[i]) - f2;
            f2[idx2] = a[i];
        }

    }
    cout<<len1<<endl<<len2;


}
```

LCS

最长公共子序列。

设有两个字符串s1， s2， 都是从 1 ~n， 1 ~ m

`dp[i][j]` 表示 对于 s1从1~i . s2从2~j， 他们的lcs 是dpi j

那么显然， 如果 s1[i] == s[j]的话， `dp[i][j] = dp[i-1][j-1]+1`

```
rep(i ,1 ,len1){
        rep(j ,1 ,len2){
            if(s[i] != t[j])f[i][j] = max(f[i-1][j] , f[i][j-1]);
            if(s[i] == t[j])f[i][j] = f[i-1][j-1]+1;
        }
    }
```

https://atcoder.jp/contests/dp/submissions/me

```
void solve(){
    string s , t;
    cin>>s>>t;
    int len1 = s.size();
    int len2 = t.size();
    s =' ' + s , t = ' ' + t;
    rep(i ,1 ,len1){
        rep(j ,1 ,len2){
            if(s[i] != t[j])f[i][j] = max(f[i-1][j] , f[i][j-1]);
            if(s[i] == t[j])f[i][j] = f[i-1][j-1]+1;
        }
```

```
    }
    //cout<<f[len1][len2]<<endl;
    stack<char>ans;
    int l = len1 , r = len2;
    while(f[l][r]){
        if(s[l] == t[r]){
            f[l][r]--;
            ans.push(s[l]);
            l-- ,r--;
        }
        else{
            if(f[l-1][r] == f[l][r])l--;
            else r--;
        }
    }
    while(!ans.empty()){
        cout<<ans.top();
        ans.pop();
    }

}
```

树形dp

考虑以为u为子树的时候， 如果u这个子节点干了， 它下面的儿子干或不干， u这个子节点没干， 它儿子干或没干

https://codeforces.com/contest/2014/problem/F

```
void dfs(int u ,int fa){
    for(auto v : g[u]){
            if(v == fa)continue;
            //f0 += f1 huo ff0
            dfs(v ,u);
            f[0][u] += max(f[0][v] , f[1][v]);
            f[1][u] += max(f[1][v] - 2ll*k  , f[0][v]);

        }
}

void solve(){
    cin>>n>>k;
    rep(i ,0 ,n+3)g[i].clear();
    memset(f , 0 ,sizeof(f));

    rep(i ,1 ,n)cin>>a[i];
    rep(i ,1,n-1){
        int u ,v;
        cin>>u>>v;
        g[u].push_back(v);
        g[v].push_back(u);
    }
    rep(i ,1 ,n)f[1][i] = a[i];

    dfs(1 , 0);
```

```
    cout<<max(f[1][1] , f[0][1])<<endl;
}
```

状压dp

https://www.luogu.com.cn/problem/P1896

所谓状态压缩就是把 每一行国王存在的方案数压缩成一个数字， 而这个数字的二进制表示，就是国王状态。

比如 5的二进制是 `00000101` ，对应的就是只有第六个和第八个位置有国王， 剩下的位置没有国王。
于是我们可以使用`f[i][j][k]`来表示放了i行，此时总共放了j个国王，第i行的国王状态为k。

```
int n , k , num;
int f[11][100][2000]; //放了i行,个了j个国王 ，第i行的状态为s的方案数量
int cnt[2000] , ok[2000]; //第i中状态的二进制中有几个1   第i个行内不相互矛盾(满足条件2：左右国王不相邻)的状态
void solve(){
  cin>>n>>k; // n *n 棋盘上面放k个国王
  for(int mask = 0;mask < (1<<n);++mask){
    int tot = 0;
    int s1 = mask;
    while(s1){
      if(s1 & 1)tot++;
      s1 >>= 1;
    }
    cnt[mask] = tot; //第i行(mask这样)的状态有tot个1
    if((((mask<<1) | (mask >>1)) & mask) == 0)ok[++num] = mask; //如果合法的话，就把这种状态存起来
  }
  f[0][0][0] = 1;

  rep(i , 1 , n){ //枚举1 ~n 行
    rep(l , 1 , num){ //枚举第i行的状态 ， 这在里我们直接枚举了所有满足条件2的状态(每一行正确的放置方法) ， 也算是个优化
      int t1 = ok[l];
      rep(r , 1 ,num){ //枚举上一行的状态
        int t2 = ok[r];
        if(  ((t2 | (t2<<1) | (t2>>1)) & t1) == 0){ //如果上下， 左上 ， 右上，都符合条件
          for(int j = 0;j<=k;++j){ //枚举国王个数
            if(j  - cnt[t1] >= 0){
              f[i][j][t1] += f[i-1][j-cnt[t1]][t2];
            }
          }
        }
      }
    }
  }
  int ans = 0;
  for(int i =1;i<=num;++i)ans += f[n][k][ok[i]];
  cout<<ans<<endl;
}
```

数位dp

> 求在给定区间[L , R]内，符合条件f(i)的数i的个数。 条件f(i)一般与数的大小无关，而和数的组成有关。 由于是按位dp， **数的大小对复杂度的影响很小**

由于数位dp状态的上下文信息比较多，所以一般用**记忆化搜索**实现，而非递推。

附上数位dp题单：https://www.luogu.com.cn/training/494976#problems

P4999 烦人的数学作业

范围很大，直接模拟会超时，于是引入数位dp的做法，一般会利用前缀和思想，把[L, R] -> [1 , R] - [1 , L-1],那么如何计算 [1 , X]?

`f[i][j]`表示从最高位开始填了i位， 数位和为j的答案

思考如何转移：因为我们从最高位开始填，那么显然每一位都有限制。拿520520举例：

- 第11位如果填0~40~4，那么后面可以随便填没有限制。
- 第11位如果填55，那么第22位就要受限，如果填0~10~1就和第一条一样，填22就是第二条，这样循环下去直到填完......

所以dfs的参数应该有三个，

- **pos**， 表示当前正在填写哪一位，从最高位`len`（原数的位数）开始往前填，根节点pos = len，即正在填最高位。 pos =0 为结束条件
- **limit**， `bool`类型的， 表示当前这一位是否有限制
- **sum**， 表示从最高位填到pos+1数位和是多少，用作递归结束的返回值

但是我们发现这样就是一个普通的模拟，把所有数都试了一遍。所以需要记忆化，如果`f[pos][sum]`已经计算过了，直接返回即可（需要注意limit == false时才能用）。

```
#include <bits/stdc++.h>
using namespace std;
#define int long long
#define rep(i ,a ,b) for(int i =a;i<=b;++i)
#define endl '\n'
#define debug(a) cout<<#a<<'='<<a<<endl;
#define inf 0x3f3f3f3f
#define ls u<<1
#define rs u<<1|1
typedef pair<int,int> pii;
int f[20][200] , a[20];
const int mod = 1e9+7;
int dfs(int pos , bool limit , int sum){
    if(pos == 0)return sum;
    if(!limit && f[pos][sum])return f[pos][sum];

    int r = limit ?a[pos]:9;
    int ans = 0;
    for(int i =0;i<=r;++i){
        ans = (ans + dfs(pos-1 , limit&&i==r , sum +i)) %mod;
        //依次枚举这一位填写什么
        //如果这一位没有限制，那么前一位也没有限制
        //如果这一位有限制， 那么只有这一位填的是a[pos]的时候，限制才能生效
    }
```

```
    if(!limit) f[pos][sum] = ans;
    return ans;
}

int calc(int x){
    int len = 0;
    while(x){
        a[++len] = x % 10;
        x /= 10;
    }
    return dfs(len , 1 , 0); //最前面也就是最后一位，肯定是有限制的

}

void solve(){
    int l , r;
    cin>>l>>r;
    cout<<(calc(r) - calc(l-1)+mod) %mod<<endl;

}
```

树形dp

树形dp，就是从子树结构的答案推得根节点的答案

https://www.luogu.com.cn/problem/P1352 著名典题： 没有上司的舞会

这里就是 两个转移求解

```
#include <bits/stdc++.h>
using namespace std;
#define int long long
#define rep(i ,a ,b) for(int i =a;i<=b;++i)
#define endl '\n'
#define debug(a) cout<<#a<<'='<<a<<endl;
#define inf 0x3f3f3f3f
#define ls u<<1
#define rs u<<1|1
typedef pair<int,int> pii;
const int N = 6e3+9;
vector<int>g[N];
int n;
int a[N];
int f[N][2];
void dfs(int u ,int fa){
    f[u][0] = 0;
    f[u][1] = a[u];
    for(auto v : g[u]){
        if(v == fa)continue;
        dfs(v , u);
        f[u][0] += max(f[v][0] , f[v][1]);
        f[u][1] += f[v][0];
```

```
    }
}

void solve(){
    cin>>n;
    rep(i ,1, n)cin>>a[i];
    rep(i , 1, n -1){
        int u ,v;
        cin>>u>>v;
        g[u].push_back(v);
        g[v].push_back(u);
    }
    dfs(1 , 0);
    cout<<max(f[1][0] , f[1][1])<<endl;
}


signed main(){
    ios::sync_with_stdio(false);
    cout.tie(0);
    cin.tie(0);
    int _ = 1;
    //cin>>_;
    while(_--)solve();
    return 0;
}
```

换根dp

对树上每个点跑树形dp。这样的话，不用换根dp一点一点跑的复杂度是 O(n^2)， 必炸。那么换根dp应运而生。 先以一个选定节点跑出来的最优解，通过另一个转移方程，就可以得出与他有关系的其他节点的答案。

区间dp

区间dp是划分区间长度， 我们需要保证区间较短的处理完之后再处理较长区间
最外层循环枚举区间长度
然后枚举左右端点， 右端点只需要用左端点算出来

```
for(int len =1;len<=n;++len){
    for(i =1;i+len-1<=n;++i){
        int j =i+len-1;
        for(int k =1;k<r;++k) //枚举分裂点
    }
}
```

期望dp

https://www.luogu.com.cn/problem/CF1265E

现在以pi表示概率

期望dp设状态： f[i] = 从1到i的期望（天数/步数/代价）

对于这道题，fi为从第一面镜子到第i面镜子都高兴的期望天数

- 第i天询问失败，从头开始， 此时概率为1-pi ,消耗天数为 f[i-1] + 1 + fi， 于是概率乘代价为(1-pi)(f[i-1]+1+fi)
- 第i天询问成功。 概率为pi , 消耗天数为f[i-1]+1 , pi(f[i-1] + 1)

综上： fi = (1-pi)(f[i-1] + 1 + fi) + pi(f[i-1] + 1)

pi只是概率， 真正的代码中还需要/100才行

树

倍增

```
int deep[N] , f[N][20];
void dfs(int u ,int fa){
    deep[u] = deep[fa] + 1;
    f[u][0] = fa; //从u点往上跳2^0步
    for(int j =1;(1<<j) <= deep[u];++j){
        f[u][j] = f[f[u][j-1]][j-1];
    }
    for(auto v : g[u]){
        if(v == fa)continue;
        dfs(v , u);
    }
}
int lca(int u ,int v){
    if(deep[u] < deep[v])swap(u , v);
    for(int j =19;j>=0;--j){
        if(deep[f[u][j]] >= deep[v]){
            u = f[u][j];
        }
    }
    if(u == v)return u;
    for(int j  =19;j>=0;--j){
        if(f[u][j] != f[v][j]){
            u = f[u][j];
            v = f[v][j];
        }
    }
    return f[u][0];
}

dfs(1 ,0);
```

LCA

```
//重儿子 ： 父节点中所有儿子中子树节点数目最多
int fa[N] , son[N] ,dep[N] , siz[N] , top[N];
void dfs1(int u , int father){
    fa[u] = father;siz[u] = 1;dep[u] = dep[father]+1;
```

```
    for(auto  v : g[u]){
        if(v == father)continue;
        dfs1(v , u);
        siz[u] += siz[v];
        if(siz[son[u]] < siz[v])son[u] = v;
    }
}

void dfs2(int u , int t){
    top[u] = t;
    if(!son[u])return;
    dfs2(son[u] , t);
    for(auto v : g[u]){
        if(v == fa[u] || v == son[u])return;
        dfs2(v , v);
    }
}

int lca(int u ,int v){
    while(top[u] != top[v]){
        if(dep[top[u]] < dep[top[v]])swap(u ,v);
        u = fa[top[u]];
    }
    return dep[u] < dep[v] ? u : v;
}
```

树的直径

一些直径的性质

- 若有多条直径， 则所有直径之间皆有公共点
- 直径的两段一定是叶子
- 树中距离某一直径端点最远的点， 至少有一个是该直径的另一个端点
- 对于树上任意一个点，与之距离最远的一个点，至少有一个直径的端点。
- 两棵树合并后的直径可能是 原来两个直径中的最大值， 或者 d1(i点在第一棵树上距离端点的最大值) + d2 + 1。三者中的最大值。
- 对于上面的d1怎么求呢？ i到直径两个端点中的max即为d1

一棵树上最长的路径叫做树的直径

用两次dfs可求出树的直径(**但是无法处理负边权， 处理负边权需要用树形dp**)

- 从任意一个点出发通过dfs对树进行一次遍历，找到最远节点p
- 从p出发，通过dfs求出最远节点q

```
void dfs(int u ,int fa){
    for(auto v : g[u]){
        if(v == fa)continue;
        dis[v] = dis[u] + 1;
        dfs(v , u);
    }
}
```

```
dfs(1 , 0); //此时的代码是树上的，不带权 从1出发dfs一遍
int ma = 0 , q;
for(int i =1;i<=n;++i){
    if(dis[i] > ma){
        ma = dis[i];
        q = i;
    }
}
rep(i ,1 ,n)dis[i] = 0;
dfs(q);
int w;
int ans =0;
for(int i =1;i<=n;++i)if(dis[i] > ans)ans = dis[i] , w = i;
cout<<ans<<endl;
```

树的重心

重心的定义是： 找到一个点，其所有的子树中最大的子树节点数最少，那么这个点就是这棵树的重心，删去重心后，生成的多棵树尽可能平衡

性质

- 树中所有点到某个点的距离和中，到重心的距离和是最小的。 如果有两个重心，那么它们的距离和一样
- 把两个树通过一条边相连得到一个新的树，那么新的树的重心在连接原来两个树的重心的路径上
- 把一个树添加或删除一个叶子，那么它的重心最多只移动一条边的距离。
- **一棵树最多有两个重心，且相邻**

方法

选择任意一个节点作为根，然后dfs。在过程中记录每个节点各个子树的大小。对于每个节点，其还存在一个上方的子树（即从假设根到当前节点），其大小为总节点数减去当前节点的总子树大小。

然后就能得到每个节点最大子树的节点数，取其中最小值即可。

整个算法只需要遍历一遍树，时间复杂度为 O(n) 。

```
int siz[N] , wgt[N]; //wgt是某个节点的最大子树大小
int n , mi = inf , ans;
void dfs(int u ,int fa){
    siz[u] = 1;
    wgt[u] = 0;
    for(auto v : g[u]){
        if(v == fa)continue;
        dfs(v ,u);
        siz[u] += siz[v];
        wgt[u] = max(wgt[u] , siz[v]);
    }
    wgt[u] = max(wgt[u] , n - siz[u]);
    if(wgt[u] < mi){
        mi = wgt[u];
        ans = u;
    }
```

```
}
```

图论

最短路

迪杰斯特拉（堆优化）

```
void dijk(int x){
    priority_queue<pii , vector<pii> , greater<pii>>q;

    q.push({0 , x});
    dis[x] = 0;
    while(!q.empty()){
        auto t = q.top();
        int u =t.second;
        q.pop();
        if(vis[u])continue;
        vis[u] = 1;
        for(auto to : g[u]){
            int v = to.first , val = to.second;
            if(dis[u] + val < dis[v]){
                dis[v] = dis[u] +val;
                q.push({dis[v] , v});
            }
        }
    }
}
```

```
struct Dijkstra{
    vector<vector<pair<int,int>>>g;
    vector<int>vis, ans , dis;
    explicit Dijkstra(vector<vector<pair<int,int>>&tmp , int st =1){
        g = tmp;
        dis.assign(g.size() , INF);
        vis.assign(g.size() , false);
        ans = bfs(st);
    }

    vector<int> bfs(int st = 1){
        priority_queue<pii , vector<pii> , greater<pii>>q;
        dis[st] = 0;
        q.push({0 , st});
        while(!q.empty()){
            auto [d , u] = q.top();
            q.pop();
            if(vis[u])continue;
            vis[u] = 1;
            for(auto [v , w] : g[u]){
                if(dis[v] > w + d){ //d就是dis[u]
                    dis[v] = w + d;
                    q.push({dis[v] , v});
```

```
                    }
                }
            }
            return dis;
        }

};

void solve(){
    Dijkstra d1(g , 1);
    Dijkstra d2(g ,n);
}
```

Floyd

```
int d[N][N];
int main(){
    cin>>n>>m;
    rep(i ,1 ,n){
        rep(j ,1 ,n){
            d[i][j] = i ==j ?0 : inf;
        }
    }
    rep(i , 1 ,m){
        int a, b , c;
        cin>>a>>b>>c;
        d[a][b] = d[b][a] = min(c , d[a][b]);
    }

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

}
```

最长路

```
int d, p , c , f  ,s;
int dis[N] , vis[N];
int cnt[N];
void spfa(int s){
    priority_queue<int,vector<int>,greater<int>>q;
    q.push(s);

    dis[s] = d;
    vis[s] = 1;
    cnt[s]++;
    while(!q.empty()){
```

```
        int u = q.top();
        q.pop();
        vis[u] = 0;
        if(++cnt[u] > c){
            cout<<"-1"<<endl;
            exit(0);
        }
        for(auto to : g[u]){
            int v = to.first;
            int val = to.second;
            if(dis[v] < dis[u] + val){
                dis[v] = dis[u] + val;
                if(!vis[v]){
                    vis[v] = 1;
                    q.push(v);
                }
            }
        }
    }
    return;
}
void solve(){
    cin>>d>>p>>c>>f>>s;
    rep(i , 1 , p){
        int u ,v;
        cin>>u>>v;
        g[u].push_back({v , d});
    }
    rep( i , 1 , f){
        int u ,v , w;
        cin>>u>>v>>w;
        g[u].push_back({v , d-w});
    }
    spfa(1);
    int ans = 0;
    for(int i =1;i<=c;++i)ans = max(ans , dis[i]);

    cout<<ans;
}
```

最短路树

所谓最短路径树 (ShortestPathTree)($ShortestPathTree$)，简称 SPTSPT，就是从一张**连通**图中，有树满足从根节点到任意点的路径都为**原图中根到任意点的最短路径**的树。

1. 最短路径树是一颗生成树，保证每一个点**联通**
2. 从**根节点**在这棵树上的任意点路径 = 原图两点之间的最短路径，即任意点i 都有leni = disi

最小生成树只是满足全图联通且**边权集**最小，而最短路径树是满足从根节点到任意点的最短路。

最短路径树的边权和 ≥ 最小生成树的边权和。

```
void dijk(int x){
```

```
    priority_queue<pii , vector<pii> , greater<pii>>q;

    q.push({0 , x});
    dis[x] = 0;
    while(!q.empty()){
        auto t = q.top();
        int u =t.second;
        q.pop();
        if(vis[u])continue;
        vis[u] = 1;
        for(auto to : g[u]){
            int v = to.first , val = to.second;
            if(dis[u] + val < dis[v]){
                dis[v] = dis[u] +val;

                q.push({dis[v] , v});
            }
        }
    }
    return
}
```

我们在进行该算法，对于每一个节点都是由一条边拉进来的，当前存入dis这个集合，每次进行松弛的时候都会将一个新点拉入集合，这样从根节点(源点)可以形成树。因为在树中**一个点就可以对应一条边**，我们可以使用一个数组pre来记录点 i 的前驱，即从源点到点i的**上一条边的编号。**

Tip: 这里 *pre* 记录是**边的编号**而不是点的编号。

而很多时候，我们需要保证树上的所有 **边权和最小** ，所以我们可以采用一个贪心思想，进行松弛的时如果**松弛前的结果** 与 **松弛后的结果** 相等即 `dis[v] = dis[u] + w` ，可以比较两种情况时，**连接这条点的边的大小**，即 w[i] 和 w[pre[next]] ， 如果 w[i] < w[pre[next]] ， 那么更新当前的pre

```
for(auto to : g[u]){
    int v = to.first , val = to.second;
    if(dis[v] > dis[u] + val){
        dis[v] = dis[u] + val;
        q.push({dis[v] , v});
        pre[v]
    }
}
```

最小生成树

```
    struct edge{
    int u ,v , w;
    bool operator <(const edge & p)const{
        return w < p.w;
    }
}e[N];
int fa[N];
int find(int x){
    return fa[x] = fa[x] == x ? x : find(fa[x]);
```

```
}

    auto kruskal = [&]()->void{
        int cnt = 0;
        rep(i , 1 , m){
            int fx = find(e[i].u) , fy = find(e[i].v);
            if(fx == fy)continue;
            fa[fx] = fy;
            cnt++;
            ans += e[i].w;

            if(cnt == n-1){

                f = true;
                break;
            }
        }
    };
```

克鲁斯卡尔重构树

Kruskal重构树—性质

1.是一个**小/大根堆**(由建树时**边权的排序方式**决定)
2.**LCA(u,v)的权值**是 **原图** u到v路径上**最大/小边权的最小/大值**(由建树时**边权的排序方式**决定)

货车运输这题是求最小边权的最大值， 于是建立 **最大生成树**， lca就是最小边权最大值

```
#include <bits/stdc++.h>
using namespace std;
#define int long long
#define rep(i ,a ,b) for(int i =a;i<=b;++i)
#define endl '\n'
#define debug(a) cout<<#a<<'='<<a<<endl;
#define inf 0x3f3f3f3f
#define ls u<<1
#define rs u<<1|1
typedef pair<int,int> pii;
//最小的最d值 -> 最大生成树
const int N = 2e4+9;
int a[N] , n , m;
vector<int>g[N];
int val[N];
int fa[N]; //并查集用
int cnt =0;
struct edge{
    int u , v , w;
    bool operator<(const edge & pre){
        return w > pre.w;
    }
}e[N];

int find(int x){
    return fa[x] = fa[x] == x ? x : find(fa[x]);
}
```

```cpp
int ff[N] , son[N] , dep[N] , siz[N] , top[N]; //ff是树链剖分用
void dfs1(int u , int father){
    ff[u] = father , siz[u] = 1;
    dep[u] = dep[father]+1;
    for(auto v : g[u]){
        if(v == father)continue;
        dfs1(v , u);
        siz[u] += siz[v];
        if(siz[son[u]] < siz[v])son[u] = v;
    }
}

void dfs2(int u  , int t){
    top[u] = t;
    if(!son[u])return;
    dfs2(son[u] , t);
    for(auto v : g[u]){
        if(v == ff[u] || v == son[u])continue;
        dfs2(v , v);
    }
}

int lca(int u , int v){
    while(top[u] != top[v]){
        if(dep[top[u]] < dep[top[v]])swap(u , v);
        u = ff[top[u]];
    }
    return dep[u] < dep[v] ?u : v;
}


void kruskal(){
    rep(i ,1 ,m){
        int fx = find(e[i].u) , fy = find(e[i].v);
        if(fx == fy)continue;
        cnt++;
        fa[fx] =cnt;
        fa[fy] = cnt;
        g[cnt].push_back(fx);
        g[cnt].push_back(fy);
        g[fx].push_back(cnt);
        g[fy].push_back(cnt);
        val[cnt] = e[i].w;
    }
}

int q;
void solve(){
    cin>>n>>m;
    rep(i ,1 , m){
        int u , v , w;
        cin>>u>>v>>w;
        e[i] = {u , v,  w};
    }
    sort(e+1 , e+1+m);
    cnt = n;
```

```
    rep(i ,1 , 2*n)fa[i] = i;
    kruskal();

    for(int i =1;i<=cnt;++i){
        if(fa[i] == i){
            dfs1(i , 0);
            dfs2(i ,i);
        }
    }


    cin>>q;
    while(q--){
        int  x , y;
        cin>>x>>y;
        if(find(x) != find(y)){
            cout<<"-1"<<endl;
            continue;
        }
        cout<<val[lca(x , y)]<<endl;

    }


}
```

负环（spfa）

```
vector<pii>g[N];
int dis[N] , vis[N];
void solve(){
    cin>>n>>m;
    rep(i , 1 , m ){
        int u ,v , w;
        cin>>u>>v>>w;
        if(w >= 0){g[u].push_back({v,w});
        g[v].push_back({u,w});
        }
        else if(w < 0)g[u].push_back({v,w});
    }
    vector<int>cnt(n+1 ,0);
    bool f= 1;
     memset(vis , 0 , sizeof(vis));
    memset(dis , 0x3f , sizeof(dis));
    auto spfa = [&](int x){
        queue<pii>q;
        q.push({x , 0});
        dis[x] = 0;
        vis[x] = 1;
        while(!q.empty()){
            auto now = q.front();
            q.pop();
            int u = now.first , val = now.second;
```

```
            vis[u] = 0;
            for(auto to : g[u]){
                int v = to.first , val = to.second;
                if(dis[v] > dis[u] + val){
                    dis[v] = dis[u] + val;
                    q.push({v , dis[v]});
                    vis[v] = 1;
                    cnt[v] = cnt[u] + 1;
                    if(cnt[v] > n){
                        f = false;
                        return;
                    }
                }
            }
        }
    };
    spfa(1);
    cout<<(f ? "NO" : "YES")<<endl;



}
```

拓扑排序

```
void toposort(){
    while(!q.empty()){
        auto u = q.front();
        q.pop();
        for(auto v :g[u]){
            inedge[v]--;
            if(inedge[v] == 0){
                dep[v] = max(dep[v] , dep[u] + 1);

            }
        }
    }



}

rep(i ,1 ,m){
    int u ,v;
    cin>>u>>v;
    g[u].push_back(v);
    inedge[v]++;
}
rep(i ,1 , n){
    if(inedge[i] == 0){
        q.push(i);
        dep[i] = 1;
    }
}
toposort();
```

```
for(int i =1;i<=n;++i)cout<<dep[i]<<endl;
```

哈密顿路

是在有向图或无向图中，恰好能将图中**所有顶点各拜访一次**的路径。

设一个无向图由N个顶点，若所有顶点的度数大于等于N/2， 则哈密顿回路一定存在 （N/2上取整）

题目：给定一张 n(n≤20) 个点的带权无向图，点从 0~n-1 标号，求起点 0 到终点 n-1 的最短Hamilton路径。 Hamilton路径的定义是从 0 到 n-1 不重不漏地经过每个点恰好一次。

思路：用二进制数字01标识当前点的状态，走过或没有走过（状压dp）

转移方程： $f[i](1 << k)[k] = min(f[i](1 >> k)[k], f[i][j] + a[j][k])$

第一维是**用一个数字的二进制**表示已经到达过了的点，第二维是当前你刚刚或者即将到达的点。

```
memset(f , 0x3f , sizeof(f));
f[1][0] = 0; //很显然第一个点是不需要花费的 ， 因为到达第一个点不需要边上的花费
for(int i =1;i<(1<<n);++i){//枚举状态
    for(int j =0;j<n;++j){ //枚举在i这种状态下的最后一个到达的点
        if((i>>j)&1){ //判断在i的状态下j点是否被用到 与(i&(1<<j))等价
            for(int k =0;k<n;++k){ //用k来更新j
                if((i^(1<<j)) >>k & 1){//如果说是点k不同于点j，而且点k在i这种状态下被经过，就说明点k可以来更新点j
                    f[i][j] = min(f[i][j] , f[i^(1<<j)][k] + a[k][j]);
                }
            }
        }
    }
}
cout<<f[(1<<n)-1][n-1]; //n-1因为图顶点从0开始
```

```
   memset(f,10,sizeof(f));
    f[1][0]=0;
    for(int i=1;i<(1<<n);i++)//复杂度2^20*400==419430400
    {
        for(int j=0;j<n;j++)//枚举你在i的状态时刚刚到达的点
        {
            if((i&(1<<j))==0)continue;//说明j这个点不在i状态所选的范围内
            for(int k=0;k<n;++k)
            {
                if((i>>k)&1)continue;//k这个点已经被选过了
                f[i|(1<<k)][k]=min(f[i|(1<<k)][k],f[i][j]+a[j][k]);//i|(1>>k)的意思是将i在二进制表示下的第k位赋值为1
            }
        }
    }
    put(f[(1<<n)-1][n-1]);//确保了起点是0，终点是n-1
```

分层图

```
int n ,m ,k ;
int st , ed;
const int N = 5e4+9;
vector<pii>g[N*11];
int dis[N*11] , vis[N*11];
void dijk(int st){
    memset(dis , 0x3f , sizeof(dis));
    priority_queue<pii , vector<pii> , greater<pii>>q;
    q.push({0 , st});
    dis[st] = 0;

    while(!q.empty()){
        auto now = q.top();
        q.pop();
        auto u = now.second;
        if(vis[u])continue;
        vis[u] = 1;
        for(auto to : g[u]){
            int v = to.first , val = to.second;
            //cout<<v<<' '<<val<<endl;
            if(dis[v] > dis[u] + val){
                dis[v] = dis[u] + val;
                q.push({dis[v] , v});
            }
        }
    }
    return;
}
void solve(){
    cin>>n>>m>>k;
    cin>>st>>ed;
    st++ , ed++;

    rep(i , 1, m){
        int u , v , w;
        cin>>u>>v>>w;
        u++ ,v++;
        g[u].push_back({v,w});
        g[v].push_back({u,w});
        rep(j , 1 ,k){
            g[u + (n*(j-1))].push_back({v + (n*j), 0});
            g[v + (n*(j-1))].push_back({u + (n*j), 0});
            g[u + (n*j)].push_back({v + (n*j), w});
            g[v + (n*j)].push_back({u + (n*j), w});
        }
    }
    rep(i , 1 ,k){
        g[ed+(n*(i-1))].push_back({ed+(n*i) , 0});
    }

    dijk(st);
    //cout<<ed<<endl;
    cout<<dis[ed+(n*k)]<<endl;
```

```
}
```

连通性

```
vector<int>g[N];
int dfn[N] , low[N];
int tot = 0;
int stk[N] , instk[N] , top;
int scc[N] , siz[N] , cnt;
void tarjan(int u){
    dfn[u] = low[u] = ++tot;
    stk[++top] = x , instk[x] = 1;
    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]);
        }
    }
    //离x时，收集scc
    if(dfn[u] == low[u]){
        int v;++cnt;
        do{
            v = stk[top--];
            instk[v] = 0;
            scc[v] = cnt;
            siz[cnt]++;
        }while(v != u);
    }
}
```

割点和割边（桥）

割边和割点的定义仅限于无向图中

割点： 无向连通图中，去掉一个点，图中的联通分量数增加
桥： 去掉一条边 ，图中的联通分量数增加

割点与桥（割边）的关系:

1）有割点不一定有桥,有桥一定存在割点

2）桥一定是割点依附的边

接下来看tarjan算法的原理:
假设dfs中我们从u到v，那么u是v的父顶点，在u之前为祖先顶点。
显然如果u的**所有孩子顶点**可以不通过父顶点就访问到祖先，说明去掉u并不影响联通性，u就不是割点

反之，如果u**至少存在一个孩子顶点**，必须通过父顶点U才能访问到U的祖先顶点，那么去掉顶点U后，顶点U的祖先顶点和孩子顶点就不连通了，说明U是一个割点。

`dnf`下表表示顶点的编号，数组的值表示该顶点在dfs中的遍历顺序（时间戳），没访问到一个未访问过的顶点，访问顺序（时间戳）的值就增加1.

`low`下标表示顶点编号， 值表示dfs中该顶点**不通过**父顶点能访问到的祖先顶点中最小的时间戳
每个顶点初始的low应该和dnf一样

割点算法

割点判定法则：

- 如果x是根节点，当搜索树上存在至少两个子节点y1 ,y2满足`low[y] >= dfn[x]`，那么x是割点
- 如果x不是根节点，只要存在x的一个子节点y，满足条件，那么x是割点

```
int n , m;
const int N = 2e4+9;
vector<int>g[N];
int dfn[N] , low[N] , tot;
int root;
bool cut[N];
void tarjan(int u){
    dfn[u] = low[u] = ++tot;
    int child = 0;
    for(auto v : g[u]){
        if(!dfn[v]){
            tarjan(v);
            low[u] = min(low[v] , low[u]);
            if(low[v] >= dfn[u]){
                child++;
                if(u != root || child > 1){
                    cut[u] = 1;
                }
            }
        }
        else{
            low[u] = min(low[u] , dfn[v]);
        }
    }
}
void solve(){
    cin>>n>>m;
    rep(i ,1 ,m){
        int u ,v ;
        cin>>u>>v;
        g[u].push_back(v);
        g[v].push_back(u);
    }
    for(root = 1;root <=n;++root){
        if(!dfn[root])tarjan(root);
    }
    int ans = 0;
    rep(i ,1 ,n)if(cut[i])ans++;
    cout<<ans<<endl;
    rep(i ,1 ,n)if(cut[i])cout<<i<<' ';
```

```
}
```

割边算法

割边判定法则：

当搜索树上存在x的一个子节点y，满足 `low[y] > dfn[x]` 那么 (x,y)这条边是割边

```
struct edge{
    int u ,v;
};
vector<int>e; //边集
vector<int>h[N]; //出边
struct bridge{int x, y;}bri[M]; //桥
int dfn[N] , low[N] ,tot ,cnt;
void add(int a ,int  b){
    e.push_back({a,b});
    h[a].push_back(e.size()-1); //边的编号从0开始
}
void tarjan(int x , int in_edg){
    dfn[x] = low[x] = ++tot;
    for(auto v : h[x]){
        //v表示的是边
        int y = e[v].v; //y表示这条边的终点
        if(!dfn[y]){
            tarjan(y , v);//把入边放进去
            low[x] = min(low[x] , low[y]);
            if(low[y] > dfn[x]){
                //割边
                bri[++cnt] = {x,y};
            }
            else if(v != (in_edg^1)){ //不是反边
                low[x] =min(low[x] , dfn[y]);
            }
        }
    }
}
```

```
#include <bits/stdc++.h>
using namespace std;
#define int long long
#define rep(i ,a ,b) for(int i =a;i<=b;++i)
#define endl '\n'
#define debug(a) cout<<#a<<'='<<a<<endl;
#define inf 0x3f3f3f3f
#define ls u<<1
#define rs u<<1|1
typedef pair<int,int> pii;
const int N = 200 , M = 5050;
int n,m;
struct edge{
    int u ,v;
};
vector<edge>e;
vector<int>h[N]; //出边
```

```
struct bri{
    int u,v;
    bool operator<(const bri & p){
        if(u == p.u){
            return v < p.v;
        }
        return u < p.u;
    }
}bri[M];
void add(int a , int b){
    e.push_back({a,b});
    h[a].push_back(e.size()-1);
}
int dfn[N] ,low[N] , tot ,cnt;
void tarjan(int x , int in_edge){
    dfn[x] = low[x] = ++tot;
    for(auto j : h[x]){
        //j是入边
        int y = e[j].v;
        if(!dfn[y]){
            tarjan(y , j);
            low[x] = min(low[x] , low[y]);
            if(low[y] > dfn[x]){
                bri[++cnt] = {x,y};
            }
        }
        else if(j != (in_edge^1)){
            low[x] = min(low[x] , dfn[y]);
        }
    }
}
void solve(){
    cin>>n>>m;
    rep(i ,1, m){
        int u,v;
        cin>>u>>v;
        add(u,v);
        add(v,u);
    }
    rep(i ,1 ,n){
        if(!dfn[i])tarjan(i , 0);
    }
    sort(bri+1 ,bri+1+cnt);
    for(int i =1;i<=cnt;++i){
        cout<<bri[i].u<<' '<<bri[i].v<<endl;
    }
}
signed main(){
    ios::sync_with_stdio(false);
    cout.tie(0);
    cin.tie(0);
    int _ = 1;
    //cin>>_;
    while(_--)solve();
    return 0;
}
```

缩点

只需要增加scc的入度和出度

缩点后变成有向无环图，并且可以直接走拓扑排序，**并且是逆序的**

```
vector<int>g[N];
int dfn[N] , low[N];
int tot = 0;
int stk[N] , instk[N] , top;
int scc[N] , siz[N] , cnt;
int din[N] , dout[N]; //scc入度 出度

void tarjan(int u){
    dfn[u] = low[u] = ++tot;
    stk[++top] = u , instk[u] = 1;
    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]);
        }
    }
    //离x时，收集scc
    if(dfn[u] == low[u]){
        int v;++cnt;
        do{
            v = stk[top--];
            instk[v] = 0;
            scc[v] = cnt;
            siz[cnt]++;
        }while(v != u);
    }
}

int main(){
    for(int x =1;i<=x;++i){
        for(auto y : g[x]){
            if(scc[y] != scc[x]){
                din[scc[y]]++;
                dout[scc[x]]++;
            }
        }
    }
}
```

双连通分量

双连通分量分为**点双连通分量** 和 **边双连通分量**

- 点双连通图： **不存在割点的无向连通图**我们称为**点双联通图。**
- 边双连通图：**不存在割边的无向连通图**我们称为**点双联通图。**
- 点双连通分量（点双）：一张图的极大点双连通子图称为**点双连通分量（V-BCC）。**
- 边双连通分量（边双）：一张图的极大边双连通子图称为**边双连通分量（E-BCC）**。
- 点双连通：若两点 u,v 在同一个点双连通分量内，那么我们称 u,v 点双连通。
- 边双联通：若两点 u,v 在同一个边双连通分量内，那么我们称 u,v 边双连通。

在一张无向图中，如果(u,v)直接相连，那么u，v点双连通，但不一定边双连通

同时，双连通分量还有一个很好的性质，我们将强连通分量缩点后可以得到一个**DAG**，但是双连通分量缩点之后可以得到**一棵树**，也就是我们在进阶中要讲的**圆方树**。

eDCC

无向图中极大的不包含割边的连通块

tarjan算法求eDCC：

- 将所有割点打标记
- 用一个栈存点，如果遍历完x发现`dfn[x] == low[x]`，说明以**x为根节点的子树中，还在栈中的节点就是连通块的节点**

整数二分

当区间变成[l ,mid] 和 [mid+1 ,r]

第一种找目标值在数组中最 左 边出现的位置

```
while(l < r){
    int mid = l+r>>1;
    if(check(mid))r = mid;
    else l =mid+1;
}
```

[l ,mid-1]和[mid ,r]

第一种找目标值在数组中最 右 边出现的位置

```
while(l < r){
    int mid = (l+r+1)>>1;
    if(check(mid))l = mid;
    else r = mid-1;
}
```

基础数据结构

ST表

ST表基于倍增思想， 可以做到**nlogn**预处理， **O(1)查询，但是不支持修改** 所以在有大量查询的问题十分有效

$f(i,j)$表示区间$[i,i+2^j-1]$的最大值

显然$f(i,0)=a[i]$ 第二维就相当于倍增的时候"跳了2^j-1步， 于是可以写出状态方程
$f(i,j)=max(f(i,j-1),f(i+2^{j-1},j-1)$

以上就是预处理部分。对于查询，可以简单实现如下：对于每个询问[l,r]，把它分成两部分：
$f[l,l+2^s-1]f[r-2^s+1,r]$
其中 $s=log2(r-l+1)$

```
int f[500001][16]
void st()
{
    for(int i =1;i<=n;++i)f[i][0] = a[i];
    for (int j = 1; j < 20; ++ j )
        for (int i = 1; i + (1 << (j - 1)) <= n; ++ i )
            f[i][j] = max(f[i][j - 1], f[i + (1 << j - 1)][j - 1]);
//Huozhe min
}
int rmq(int i , int j){ //查询：返回区间[i,j]的最值
    int k = log(double(j-i+1)) / log(2.0);
    return min(f[i][k], f[j - (1 << k) + 1][k]);
}
```

https://atcoder.jp/contests/abc388/tasks/abc388_g

```
#include <bits/stdc++.h>
using namespace std;
#define int long long
#define rep(i ,a ,b) for(int i =a;i<=b;++i)
#define endl '\n'
#define debug(a) cout<<#a<<'='<<a<<endl;
#define inf 0x3f3f3f3f
#define ls u<<1
#define rs u<<1|1
typedef pair<int,int> pii;
const int N = 2e5+9;
int a[N];
int f[N][20];
int n;
int rmq(int i, int j)
{
    int k = floor(log((double)(j - i + 1)) / log(2.0));
    return max(f[i][k], f[j - (1 << k) + 1][k]);
}
bool check(int x , int L , int R){
    int ll  =L;
    int rr = L + x-1;
```

```
    int ma = rmq(ll , rr);
    int tmp = R - x - L+1;
    return ma <= tmp;
}
void st()
{

    for (int j = 1; j < 20; ++ j )
        for (int i = 1; i + (1 << (j - 1)) <= n; ++ i )
            f[i][j] = max(f[i][j - 1], f[i + (1 << j - 1)][j - 1]);
}

void solve(){
 cin>>n;
 rep(i ,1 , n)cin>>a[i];

   for(int i=1;i<=n;i++)
    {
        int j=lower_bound(a+1,a+n+1,a[i]*2)-a;
        f[i][0]=j-i;
    }

    st();
    int q;cin>>q;
    while(q--){
        int L ,R;
        cin>>L>>R;
        int l = 0 , r = (R-L+1) / 2;
        while(l < r){
            int mid = (l+r+1)>>1;
            if(check(mid , L , R))l = mid;
            else r = mid-1;
        }
        cout<<l<<endl;
    }
}
```

单调队列

deque不具有单调性， 是人为写出来的

滑动窗口

```
#include <bits/stdc++.h>
using namespace std;
#define int long long
#define rep(i ,a ,b) for(int i =a;i<=b;++i)
#define endl '\n'
#define debug(a) cout<<#a<<'='<<a<<endl;
#define inf 0x3f3f3f3f
#define ls u<<1
#define rs u<<1|1
typedef pair<int,int> pii;
const int N =1e6+9;
int a[N];
```

```
int n , k;
deque<int>q1 ,q2;
void solve(){
    cin>>n>>k;
    rep(i , 1 ,n)cin>>a[i];
    rep(i ,1 ,n){
        while(!q1.empty() && q1.front() <=i-k)q1.pop_front();
        while(!q1.empty() && a[q1.back()] > a[i])q1.pop_back(); //最后一个一定是最大
的 从小到大
        q1.push_back(i);
        if(i>=k)cout<<a[q1.front()]<<' ';
    }
    cout<<endl;

    rep(i ,1 ,n){
        while(!q2.empty() && q2.front() <=i -k)q2.pop_front();
        while(!q2.empty() && a[q2.back()] < a[i])q2.pop_back();  //从大到小
        q2.push_back(i);
        if(i >=k)cout<<a[q2.front()]<<' ';
    }

}


signed main(){
    ios::sync_with_stdio(false);
    cout.tie(0);
    cin.tie(0);
    int _ = 1;
    //cin>>_;
    while(_--)solve();
    return 0;
}
```

单调栈

单调栈是一种特殊的栈， 栈内元素始终保持单调有序。
应用场景： 求解某个元素的左边或右边第一个比它大或小的元素

模板题：https://www.luogu.com.cn/problem/P5788
求解每个人右边第一个比他高的人的下标

因为是右边（应该是这样，不确定）， 所以倒着排序，比他高，就维护一个单调递减的栈

```
void solve(){
    cin>>n;

    for(int i =1;i<=n;++i)cin>>a[i];
    stack<int>st;
    vector<int>ans(n+1);

    for(int i =n;i>=1;--i){
        while(!st.empty() && a[i] >= a[st.top()])st.pop();
```

```
        if(st.empty())ans[i] = 0;
        else ans[i] = st.top();
        st.push(i);
    }
    for(int i =1;i<=n;++i)cout<<ans[i]<<' ';
}
```

并查集

路径压缩

```
int find(int x){
    return fa[x] = fa[x] == x ? x : find(fa[x]);
}
```

启发式合并(nlogn)

```
void unionn(int x , int y){
    x = find(x) , y = find(y);
    if(siz[x] > siz[y])swap(x , y); //x为小的值
    fa[x] = y;
    siz[y] +=siz[x];
}
```

带权并查集

增加了一个权值之后，我们就需要考虑两个问题：

- 每个节点都记录的是与根节点之间的权值，那么在find的路径压缩过程中，权值也应该做相应的更新，因为在路径压缩之前，每个节点都是与其父节点链接着，那个Value自然也是与其父节点之间的权值
- 两个并查集合并时，权值也要做相应的更新，因为两个并查集的根节点不同。

向量偏移法 路径压缩

查询操作，只需要在普通并查集的基础上更新value即可——x到t(t = fa[x])的距离加t到根节点的距离 就是 x到跟根节点的距离

```
int find(int x){
    if(x == fa[x])return x;
    int t = fa[x]; //记录原本的父节点
    fa[x] = find(fa[x]);//父节点变成了根节点 ， 此时val[x] = 父节点到根节点的权值
    val[x] += val[t]; //当前节点的权值加上原本父节点的权值
    reuturn fa[x];
}
```

路径压缩后， 父节点直接变成了根节点， 此时父节点的权值已经是父节点到根节点的权值了， 将当前节点的权值加上原本父节点的权值，就得到当前节点到根节点的权值

合并

```
int fx = findSet(x);
int fy = findSet(y);
if(fx != fy){
    fa[fx] = fy;
    val[fx] = -val[x] + val[y] + s;
}
```

例题： 银河英雄传说

```

const int N =30000;
int fa[N] , val[N] , s[N];
int find(int x){
    if(x == fa[x])return x;
    int t = fa[x];
    fa[x] = find(fa[x]);
    val[x] += val[t];
    return fa[x];
}
void solve(){
    int n = N;
    int t ;cin>>t;
    rep(i ,1 ,n)fa[i] = i , val[i] =0 ,s[i] = 1;
    rep(i ,1, t){
        char op;
        cin>>op;
        int x, y;
            cin>>x>>y;
            int fx = find(x) , fy = find(y);
        if(op == 'M'){

            fa[fx] = fy; //x到y后面
            val[fx] += s[fy];
            s[fy] += s[fx];
            s[fx] = 0;
        }
        else{
            if(fx == fy){
               cout<<abs(val[x] - val[y])-1<<endl;
            }
            else cout<<"-1"<<endl;
        }
    }


}
```

val[x]是x到根节点的距离， 每次 find()都会路径压缩更新新的根节点

扩展域并查集

n个点有m对关系，把n个节点放入两个集合中，要求每对存在关系的两个节点不能放在同一个集合中。

```
void solve(){
  cin>>n>>m;
  rep(i , 1 , m)cin>>a[i].l>>a[i].r>>a[i].w;
  sort(a+1 , a+1+m);
    rep(i , 1 ,2*n)fa[i] = i; //精华

    rep(i , 1 , m){
        int fx = find(a[i].l) , fy = find(a[i].r);
        if(fx == fy){
            cout<<a[i].w<<endl;
            return;
        }
        merge(a[i].l, a[i].r+n); //精华 ，  剩下不变
        merge(a[i].l+n , a[i].r);//精华
    }
    cout<<0<<endl;

}
```

树状数组

- 数组前缀和查询
- **单点**修改k （**这里k表示ai + k**）
- **区间修改成k**
- 可以维护多个数组之间的相对位置(2024湖南省赛)

```
struct BIT{
    int n = N;
    int tr[N];

    int query(int x){
        int s = 0;
        for(;x;x -=(x & -x))s +=tr[x];
        return s;
    }

    int query(int l , int r){
        return query(r) - query(l-1);
    }

    void modify(int x , int k){
        for(;x<=n;x +=(x&-x))tr[x] +=k;
    }

     void modify(int l ,int r , int k){
        modify(l , k);
        if(r +1 <=n)modify(r+1 , -k);
    }
}F;
```

```
//+数字
//全局+ num
//查找 > num的
const int N = 2e5+9;
struct BIT{
    int n = N;
    int tr[N];
    int query(int x){
        int s = 0;
        for(;x;x -=(x & -x))s +=tr[x];
        return s;
    }
    int query(int l , int r){
        return query(r) - query(l-1);
    }
    void modify(int x , int k){
        for(;x<=n;x +=(x&-x))tr[x] +=k;
    }
     void modify(int l ,int r , int k){
        modify(l , k);
        if(r +1 <=n)modify(r+1 , -k);
    }
}F;
void solve(){
    int q;cin>>q;

    int cnt = 0;
    int st = 1;
    while(q--){
        int op;
        cin>>op;
        if(op == 1){
            cnt++;

            F.modify(cnt , 0);
        }
        else if(op == 2){
            int x;cin>>x;

            F.modify(1 , cnt ,x);
        }
        else{
            int h;cin>>h;

            int l = st-1 , r = cnt+1;
            while(l < r){
                int mid = l+r+1>>1;
                if(F.query(mid) >= h)l = mid;
                else r = mid-1;
            }

            cout<<l-st+1<<endl;
            st = l+1;
```

```
        }
    }
}
```

线段树

无懒标记

没有懒标记的线段树可以实现区间求和， 点单修改，区间查询

```
#define ls u<<1
#define rs u<<1|1
struct Tree{
    int l ,r , sum;
}tr[N*4];

void pushup(int u){ //上传
    tr[u].sum = tr[ls].sum + tr[rs].sum;
}
void build(int u ,int l ,int r){
    tr[u].l = l , tr[u].r = r;
    if(l == r){
        tr[u].sum = a[l];
        return;
    }
    int mid = l + r>>1;
    build(ls ,l , mid);
    build(rs ,mid+1 , r);
    pushup(u);
}
void change(int u ,int x , int k){//点修
    if(tr[u].l == x && tr[u].r == x){
        tr[u].sum += k;
        return;
    }
    int mid = tr[u].l + tr[u].r >>1;
    if(x <= mid)change(ls ,x ,k);
    if(x > mid)change(rs , x ,k);
    pushup(u);
}
int query(int u ,int l ,int r){//区间查询 要查的区间应该大
    if(l > tr[u].r || r < tr[u].l)return 0;
    if(l <= tr[u].l && tr[u].r <= r)return tr[u].sum;
    return query(ls,l,r) + query(rs,l,r);
}
```

有赖标记

有懒标记，可以实现区间修改， 区间查询

```
struct Tree{
    int l , r , sum , add;
}tr[4*N];

void pushup(int u){
    tr[u].sum = tr[ls].sum + tr[rs].sum;
}
void pushdown(int u){
    if(tr[u].add){
        tr[ls].sum += tr[u].add * (tr[ls].r - tr[ls].l + 1);
        tr[rs].sum += tr[u].add * (tr[rs].r - tr[rs].l + 1);
        tr[ls].add += tr[u].add;
        tr[rs].add += tr[u].add;
        tr[u].add = 0;
    }
}
void build(int u , int l , int r){
    tr[u].l = l , tr[u].r =r;
    if(l == r){
        tr[u].sum =w[l];
        return;
    }
    //tr[u] = {l , r ,w[l] , 0};
    //if(l == r)return;
    int mid = l+r>>1;
    build(ls , l ,mid);
    build(rs ,mid+1 , r);
    pushup(u);
}
void change(int u ,int l , int r , int k){
    if(l <= tr[u].l && tr[u].r <= r){
        tr[u].sum +=k*(tr[u].r - tr[u].l + 1);
        tr[u].add += k;
        return;
    }
    pushdown(u);
    int mid = tr[u].l + tr[u].r>>1;
    if(l <= mid)change(ls ,l , r , k);
    if(r > mid )change(rs , l , r ,k);
    pushup(u);
}
int query(int u ,int l , int r){
    if(l <= tr[u].l && tr[u].r <= r)return tr[u].sum;
    int mid = tr[u].l + tr[u].r>>1;
    pushdown(u);
    int sum = 0;
    if(l <= mid)sum += query(ls , l ,r);
    if(r > mid)sum += query(rs , l , r);
    return sum;
}
```

区间gcd

不支持修改

```
struct Tree{
    int l ,r;
    int res;
}tr[N<<2];
int gcd(int a , int b){
    return b == 0 ? a : gcd(b , a % b);
}
void pushup(int u){
    tr[u].res = gcd(tr[ls].res , tr[rs].res);
}
void build(int u , int l ,int r){
    tr[u] = {l , r , a[l]};
    if(l == r)return;
    int mid = l+r>>1;
    build(ls , 1 , mid);
    build(rs ,mid+1 ,r);
    pushup(u);
}

int query(int u , int l, int r){
    if(r < tr[u].l || l > tr[u].r)return 0;
    if(l <= tr[u].l && tr[u].r <=r)return tr[u].res;
    int mid  = tr[u].l + tr[u].r>>1;
    int lres = query(ls ,l , r);
    int rres = query(rs ,l ,r);
    return gcd(lres , rres);
}
```

权值线段树

普通线段树维护的是数列的区间信息，比如区间最大值，区间最小值等等。在维护序列的这些信息的时候，我们更关注的是这些数本身的信息，换句话说，我们要维护区间的最值或和，我们最关注的是这些数统共的信息。而权值线段树**维护一列数中数的个数**。

比如说 `1 1 1 2 3 3 4 5 5 6 6 7`一棵权值线段树的叶子节点维护的是“有几个1”，“有几个2”...，他们的父亲节点维护的是“有几个1和2”。
也就是维护一个桶。权值线段树上表示区间 [1,1][1,1] 的节点值为 33 表示有 33 个 11 ， 表示区间 [2,2][2,2] 的节点值为 11 ， 而他们的父节点表示 [1,2][1,2] ， 值为 44 ， 表示有 44 个 11 和 22 ，以此类推。

权值线段树可以解决数列的 **第k大问题** ， 因为我们的权值线段树维护的是一堆桶，每个节点储存的是节点维护区间（就是值域）的数出现的次数（再次强调定义），那么，根据这个定义，整棵线段树的根节点就表示整个值域有几个数。对于整列数中第k大/小的值，我们从根节点开始判断（这里按第k大为例），如果k比右儿子大，就说明第k大的数在左儿子所表示的值域中。然后，k要减去右儿子。（这是因为继续递归下去的时候，正确答案，整个区间的第k大已经不再是左儿子表示区间的第k大了，很好理解）。

建树

```
void build(int u , int l  , int r){
    if(l == r){
        tr[u] = a[l]; //a[l]有多少个
        return;
    }
    int mid = (l+r)>>1;
    build(ls , l , mid);
    build(rs , mid+1 , r);
    pushup(u);
}
```

修改

```
void update(int u,int l,int r,int k,int cnt)//表示数k的个数多cnt个
{
    int mid=(l+r)>>1;
    if(l==r)
    {
        tree[u]+=cnt;
        return;
    }
    if(k<=mid)
        update(ls,l,mid,k,cnt);
    else
        update(rs,mid+1,r,k,cnt);
    pushup(u);
}
```

查询

```
int query(int u ,int l ,int r , int k) //查询数k有多少个
{
    int mid = l+r>>1;
    if(l == r){
        return tr[u];
    }
    if(k <= mid){
        return query(ls ,l ,mid , k);
    }
    else return query(rs , mid+1 , r , k);
}
```

查询第k大/小值

```
int kth(int u , int l ,int r , int k)//查询第k大值是多少
{
    int mid =l+r>>1;
    if(l == r)return l;
    if(k < mid)return kth(rs , mid+1 , r ,k);
    else return kth(ls , l ,mid , k -m)

}
```

数论

一些符号

整除： 若非0整数a是整数b的因数 即： b mod a == 0， 则称 a整除b 或b被a整除 记作 a | b

同余方程

`(a + b)mod x ==  0  - > (a modx + b mod x) mod x = 0`

`(a-b)mod x == 0 -> a mod x - b mod x == 0`

余数的公式： $N\%i = N - \lfloor \frac{N}{i} \rfloor * i$

a %b =k -> a-b是k的倍数

鸽巢原理

一种表达是这样的：如果要把n个对象分配到m个容器中，必有至少一个容器容纳至少$\left\lceil \frac{n}{m} \right\rceil$个对象。

置换环

结论： 排序所需的最小交换次数 = 数组长度 - 环的个数

gcd和lcm

gcd ->最小公倍数

```
int gcd(int a , int b){
    return b == 0 ? a : gcd(b , a % b);
}
```

快速幂

a的b次方

```
int ksm(int a , int b){
    int res = 1;
    while(b){
        if(b&1) res = res * a % mod;
        b >>= 1;
        a = a * a % mod;
    }
    return res% mod;
}
```

```
int ksm(int a , int b){
    int res = 1;
    while(b){
        if(b&1) res = res * a;
        b >>= 1;
        a = a * a;
    }
    return res;
}
```

逆元

inv(a)可以看作模p意义下的$\frac{1}{a}$,那么在取模意义下， $\frac{a}{b}$就变形为 `a*inv(b)(mod p)`

费马小定理求逆元

只对p是质数的情形有效

```
int inv(int b){
    return ksm(b , mod-2) %  mod;
}
```

组合数

```
int ksm(int a , int b){
  int res = 1;
  while (b > 0) {
    if (b & 1) res = (res * a)%mod;
    a = (a * a)%mod;
    b >>= 1;
  }
  return res%mod;
}
int C(int n , int m){
    return pre[n] * ksm(pre[m]*pre[n-m]%mod , mod-2)%mod;
}
int C(int n ,int m){
    return pre[n] * inv(pre[m] * pre[n-m] %mod) % mod;
}
```

```
pre[0] = 1;
rep(i , 1 , N)pre[i] = (pre[i-1] * i) % mod;
```

裴蜀定理

a， b为常数

```
ax + by =m 有解  ax +by = k * gcd(a,b);
```

数论分块

余数的公式： $N\%i = N - \lfloor \frac{N}{i} \rfloor * i$
数论分块，通常用于快速求解形如

的和式，所以通常被称为 **整除分块**，当能用 O(1)O(1) 计算出 r∑i=lf(i)∑i=lrf(i) 时，数论分块便能用 O(√n)O(n) 的时间计算出上式的值、数论分块经常搭配 **莫比乌斯反演** 一起使用。

我们很显然发现$\lfloor \frac{n}{i} \rfloor$是成段出现的， 例如n=10的时候
`10 5 3 2 2 1 1 1 1 1`

```
for(int l = 1, r; l <= n; l = r + 1) {
    r = n / (n / l);
    ans += (r - l + 1) * (n / l)
}
```

分块的时间复杂度是$\sqrt{n}$

$N\%i = N - \lfloor iN \rfloor \cdot i$，其中$\lfloor Ni \rfloor$ $\lfloor \frac{N}{i} \rfloor$ $\lfloor iN \rfloor$可以使用一种叫做整除分块/数论分块的技巧暴力枚举

因数个数定理

解质因数结果为 $x = p1^{k1}p2^{k2}p3^{k3}....$，令f(x)=(k1+1)(k2+1)···(kn+1)
实际上f(x)的值就是x的因数的个数

欧拉筛

筛选前N个数中 有哪些是素数，复杂度O(n),极其优秀

所采用的一个惯用思路是：找到一个素数后，就将它的倍数标记为合数，也就是把它的倍数“筛掉”；如果一个数没有被比它小的素数“筛掉”，那它就是素数。欧拉筛法的大致思路也是如此，就是其中有些细节有差异。欧拉筛法拥有线性的复杂度，而且编码较简单，应用十分广泛。

```
bool isprime[N]; //isprime表示i是不是素数
int prime[N]; //现在以及筛出的素数列表
int n; //上限 ，即筛出<=n的素数
int cnt; //以及筛出的素数个数
```

```
void euler(){
    memset(isprime , true , sizeof(isprime));//先全部标记为素数
    isprime[1] = false;//1不是素数
    for(int i =2;i<=n;++i){
        if(isprime[i])prime[++cnt] = i;
        for(int j=1;j<=cnt && i*prime[j]<=n;++j){
            isprime[i*prime[j]] = false;
            if(i % prime[j] == 0)break;
        }
    }
}
```

```
struct Ola_prime{
    vector<int>prim , minp;
    void init(int _n = 10000005){ //筛1~_n的质数
        prim.clear();
        minp.assign(_n+5 , 0);
        for(int i =2;i<=_n;++i){
            if(minp[i] ==0)prim.push_back(i) , minp[i] = i;
            for(int x : prim){
                if(i*x >_n)break;
                minp[i*x] = x;
                if(x == minp[i])break;
            }
        }
    }
    bool isprim(int x){ //x是不是质数，是1
        return minp[x] == x;
    }
    int minprim(int x){ //返回x的最小质因数
        return minp[x];
    }
    int kth(int k ){ //返回第k个质数
        return prim[k-1];
    }
};
Ola_prime P;
```

莫比乌斯反演(实在太超标)

$n \perp m$ 是指 n 与 m 互质（注意，$1 \perp 1$ 是成立的）

数论函数 & 迪利克雷卷积

数论函数： 其定义域是正整数， 值域是一个数集

定义两个数论函数的加法， 为逐项相加，即(f+g)(n) = f(n) +g(n)
数乘（一个数乘到一个数论函数上），(xf)(n) = x * f(n)

狄利克雷卷积
两个数论函数的迪利克雷卷积:
t = f * g
$t(n) = ij = n \sum f(i)g(j)$

- 若 $n \perp m$ 则每个 nm 的约数都可以分解成一个 n 的约数和一个 m 的约数的积
- 若 $n \perp m, a \mid n, b \mid m$ 则 $a \perp b$ 。

字符串

字典树

`Trie`是一种能够快速插入和查询字符串的多叉树结构

Trie维护字符串的集合，支持两种操作

- 向集合中插入一个字符串，void insert(char * s)
- 查询字符串

建字典树

儿子数组 ch[p] [j]：存储从节点P 沿着 j 这条边走到的子节点。边为26个小写字母，每个节点最多可以有26个分叉

ch[0] [2]从0号节点沿着 c 字母这儿跳变走到的节点是2号节点

计数数组 cnt[p] : 存储以节点p结尾的单词的插入次数
节点编号：用来给节点编号

1. 空Trie仅有一个根节点， 编号为0
2. 从根开始查，枚举字符串的每个字符， 如果有儿子，p指针走到儿子， 如果没有，先创建儿子，p指针再走到儿子
3. 在单词结束点记录插入次数

```
char s[N];
int ch[N][26] , cnt[N] , idx;

void insert(char * s){
    int p = 0;
    for(int i =0;s[i];++i){
        int j = s[i] - 'a';
        if(!ch[p][j])ch[p][j]=++idx;  //不存在这个儿子， 就创建出来
        p = ch[p][j];
    }
    cnt[p]++;
}
```

查询

1. 从根开始查，扫描字符串
2. 有字母s[i] , 则走袭来，能走到此为，则返回插入次数

3. 无字母s[i]， 返回0

```
int query(char * s){
    int  p = 0;
    for(int i =0;s[i];++i){
        int j =s[i] - 'a';
        if(!ch[p][j])return 0;
        p = ch[p][j];
    }
    return cnt[p];
}
```

01Trie

```
const int N = 2e5+9;
int ch[N*31][2] , idx;
int n ,a[N];

void insert(int x){
    int p =0;
    for(int i = 30;i>=0;--i){
        int j = x >>i & 1;
        if(!ch[p][j])ch[p][j] = ++idx;
        p = ch[p][j];
    }
}

int query(int  x){
    int p =0 , res = 0;
    for(int i =30;i>=0;--i){
        int j =x >> i & 1;
        if(ch[p][!j]){
            res += 1<<i;
            p = ch[p][!j];
        }
        else p = ch[p][j];
    }
    return res;
}

void init() {
    rep(i, 0, idx) { ch[i][0] = ch[i][1] = 0; }
    idx = 1;
    insert(0);
}

int main(){
    cin>>n;
    rep(i ,1 ,n)cin>>a[i] , insert(a[i]);
    int ans =0;
    rep(i , 1 , n)ans = max(ans , query(a[i]));
}
```

注意： 同或的时候， 会选择自己同或

杂

二进制枚举

```
int n;cin>>n;
for(int i =0;i<(1<<n);++i){
    for(int j =0;j<n;++j){ //遍历二进制的每一位
        if(i&(1<<j)){ //判断二进制的第j位是否存在
            cout<<j<<endl;
        }
    }
}
```

异或和的性质

f(n)表示从1到n的前缀异或和，那么

`f(n) = n   n mod 4 == 0`

`f(n) = 1  n mod 4 == 1`

`f(n) =n+1  n mod 4 == 2`

`f(n) = 0 n mod 4 == 3`

区间异或和： [l , r] = pre[r] ^ pre[l-1];

https://codeforces.com/contest/2036/problem/F

快速或

从l或到r

```
慢速为On
for(int i =l;i<=r;++i) ....
快速
while(l < r){
    l |= l+1;
}
```

二维偏序

形如 xi < xj 且 yi < yj 之类的 约束条件，我们可以称为二维偏序。

逆序对就是一个经典的二维偏序。

如果按照暴力想法，我们 O(n2)O(n2) 的时间枚举 i,ji,j ,这样太慢了。

处理第 ii 位时，我们已经处理过 [0,i−1][0,i−1] 的数量，那么我们可不可以用一个数据结构记录一下之前的情况呢?

这就引出了二维偏序。

我们把第一维从小到大排序，然后遍历，将第二位插入树状数组中，每次查询，即可解决问题。

首先我们来看逆序对

```
const int N = 2e5+9;
struct BIT{
    int n = N;
    int tr[N];

    int query(int x){
        int s =0;
        for(;x;x -=(x&-x))s += tr[x];
        return s;
    }
    int qeury(int l ,int r){
        return query(r) - query(l-1);
    }

    void modify(int x ,int k){
        for(;x <=n;x+=(x&-x))tr[x] += k;
    }

    void modify(int l ,int r , int k){
        modify(l ,k);
        if(r +1 <=n)modify(r+1 , -k);
    }
}F;
int n;
int a[N];
void solve(){
    cin>>n;
    int ans =0;
    int ma = 0;
    rep(i ,1 ,n)cin>>a[i] , ma = max(ma , a[i]);
    rep(i ,1 ,n){
        ans += F.query(ma) - F.query(a[i]);
        F.modify(a[i] , 1);
    }
    cout<<ans<<endl;
}
```

洛谷中会re，因为需要离散化

```
struct BIT{
    int n = N;
    int tr[N];

    int query(int x){
        int s =0;
```

```
        for(;x;x -=(x&-x))s += tr[x];
        return s;
    }
    int qeury(int l ,int r){
        return query(r) - query(l-1);
    }

    void modify(int x ,int k){
        for(;x <=n;x+=(x&-x))tr[x] += k;
    }

    void modify(int l ,int r , int k){
        modify(l ,k);
        if(r +1 <=n)modify(r+1 , -k);
    }
}F;
int n;
vector<int>num;
int a[N];
//离散化数组， 可以使用vector
void solve(){
    cin>>n;
    int ans =0;
    int ma = 0;
    rep(i ,1 ,n){
        cin>>a[i];
        num.push_back(a[i]);
    }
    sort(num.begin() , num.end());
    num.erase(unique(num.begin() , num.end()) , num.end());
    int m = num.size(); //获得去重后数组大小，即不同的数有多少个
    rep(i ,1 ,n){
        //确定a[i]在去重后升序排列的数组中的位置  ，因为树状数组从1开始存
        a[i] = lower_bound(num.begin() , num.end() , a[i]) - num.begin() + 1;
        ans += F.query(m) - F.query(a[i]);
        F.modify(a[i] , 1);
    }
    cout<<ans<<endl;
}
```

https://ac.nowcoder.com/acm/contest/16/A

```
const int N = 1e5+9;
struct BIT{
    int tr[N];
    int n =N;
    int query(int x){
        int s =0;
        for(;x;x -=(x&-x))s += tr[x];
        return s;
    }
    void modify(int x){
        for(;x<=n;x+=(x&-x))tr[x] +=1;
    }
}F;
```

```
struct node{
    int m ,v; //内存和速度
    bool operator<(const node & pre){
        return v < pre.v;
    }
}a[N];
int n;
vector<int>v;
void solve(){
    cin>>n;
    rep(i ,1 ,n){
        cin>>a[i].m>>a[i].v;
        v.push_back(a[i].m); //对内存排序
    }
    sort(v.begin() , v.end());
    v.erase(unique(v.begin() , v.end()) , v.end());
    for(int i =1;i<=n;++i){
        a[i].m = lower_bound(v.begin() , v.end() , a[i].m)-v.begin()+1;
    }
    int ans=0;
    sort(a+1 , a+1+n);
    for(int i =n;i>=1;--i){
        int cnt = F.query(a[i].m); //找内存比当前大并且速度比当前小的有几个
        if(n -i -cnt !=0)ans++; //被完虐
        F.modify(a[i].m);
    }
    cout<<ans<<endl;

}
```

trick

把数组初始化成-1， 然后 (~f[])就代表赋值过了，在记忆化搜索可用

快速计算出所有的a[j] - a[i]之和(j > i)

```
先排序sort
先计算出前缀和
for(int i =2;i<=n;++i){
            int t = (a[i] * (i-1)- pre[i-1]);
            s +=t;
}
cout<<s<<endl;
```

a^b = a + b

此时 a & b == 0

梅森旋转随机数

```
std::mt19937_64 rnd(std::time(0));
for(int i =1;i<=n;++i)a[i] = rnd();
```