ACM

基础知识

卡常

```cpp
#include<bits/stdc++.h>
#include<stime>
using namespace std;
int start = clock();
//由于clock函数本身非常慢，所以可以每个10000次使用一次clock函数来卡常，
int main()
{
    for(int i=0;;i++)
    {
        if(i % 10000 == 0 && clock() - start >= 850000)// 1/1000000 s
        {
            cout << i << endl;
            exit(0);
        }
        /*或者另一种写法*/
        if(i % 10000 == 0 && clock() - start >= CLOCKS_PER_SEC *0.9)
        {//CLOCKS_PER_SEC变量名 值为 1/1000000 s
            cout << i << endl;
            exit(0);
        }
    }
    return 0;
}
```

STL 二分

```
这两个二分查找操作可以在set，数组，vector，map中使用：

数组 或者 vector 中的语法：
序列是升序的（从小到大）
lower_bound(begin(),end(),x) //返回序列中第一个大于等于x的元素的地址
upper_bound(begin(),end(),x) //返回序列中第一个大于x的元素的地址

序列是降序的（从大到小）
lower_bound(begin(),end(),x,greater<tpye>()) //返回序列中第一个小于等于x的元素的地址
upper_bound(begin(),end(),x,greater<type>()) //返回序列中第一个小于x的元素的地址

set 或者 map 中的语法：
和数组差不多，只不过返回的是迭代器：
s.lower_bound(x) //返回序列中第一个大于等于x的元素的地址
s.upper_bound(x) //返回序列中第一个大于x的元素的地址

std::binary_search(ans + 1, ans + ct + 1, h[a][b]) ? "1" : "0");
//返回类型是bool型,即是,只能返回真或假,不能返回存在位置的下标
重点注意：如果当前序列中找不到符合条件的元素，那么返回end(),对于数组来说，返回查询区间的首地址位置，对于set来讲，返回end()-1后面元素的迭代器，也就是begin();
```

vector

> - vector 数组查找　if (find(res.begin(),res.end(),b.substr(i, 2))!=res.end()) // //vector数组find函数，从begin开始到end结束，后面的是查找的内容。
> - vector 数组去重

```cpp
//set 数组辅助去重
set<int>s(vec.begin(), vec.end());
vec.assign(s.begin(), s.end());
//vector 使用unique函数去重，这个函数只能去重相邻的数据
sort(vec.begin(),vec.end());
//vector <int>::iterator it;
//it=unique(vec.begin(),vec.end());
//vec.erase(it,vec.end());
vec.erase(unique(vec.begin(), vec.end()), vec.end());
```

- vector初始化方式

```
//使用vector一次性完成二维数组的定义（注意：此种方法适用于每一行的列数相等的二维数组）
vector<vector<int>> matrix(m, vector<int>(n, -1));
//以下是拆分理解
//创建一维数组matirx，这个数组里有m个元素，元素是int型vector。
vector<vector<int>> matrix(m);
//除了定义数组类型及数组大小外，同时给数组中的元素赋值：将元素赋值为大小为n的int型vector。
vector<vector<int>> matrix(m, vector<int>(n));
//除了定义数组类型、数组大小、列的大小，同时给数组列中的元素（或者说，数组中的所有元素）赋值为-1。
vector<vector<int>> matrix(m, vector<int>(n, -1));
//当每一行的列数不相等的时候
vector<vector<int>> matrix;//创建一维数组matirx，这个数组里的元素是int型vector。
int m = 3; //matrix有m行
int n = 10; //matrix有n列
int value = 1; //最终matrix成为二维数组后，其中每个元素的值为1（如果不需要进行初始化，此语句可以省略）
for (int i = 0; i < m; ++ i) {
    vector<int> tmp(n, value); //定义int型一维数组tmp，该数组有n个int型元素，且每个元素的初始值为value
    matrix.push_back(tmp); //将一维数组tmp（小容器）加入matrix（大容器）中，使之成为matrix的元素，令matrix成为二维数组
}
```

map

> - map数组可以使用 auto 的方式遍历

```
map/multimap
(它们都是关联容器，增删效率为log级别，并且依据key能自动排序，默认小于，前者key不允许重复，后者允许)
    insert()  插入的数是一个pair
    erase()  输入的参数是pair或者迭代器
    find()
    []  注意multimap不支持此操作。 时间复杂度是 O(logn)
    lower_bound()/upper_bound()
```

数学知识

> - *_lg(tmp),内置函数,返回的值是2 ^ x <= tmp ,x的最大值,即* _lg(4) = 2, __lg(5) = 2, __lg(9) = 3
> - 求n个数的最小公倍数就是求这些数中每一个同样质因子的最大指数
>   拓展：求n个数的最大公约数就是这些数中每一个同样质因子的最小指数）

迭代器

```
vector<type>::iterator iter;//type 类型
map<type,type>::iterator iter;//迭代器类型可以使用auto
set<type>::iterator iter;
等等.....
迭代器可以像指针一样，遍历STL时可以直接对迭代器 ++ --  ；
访问迭代器的值的形式：
*iter
iter->first    iter->second
```

int128

```
inline __int128 read(){
    __int128 x=0,f=1;
    char ch=getchar();
    while(ch<'0'||ch>'9'){
        if(ch=='-')
            f=-1;
        ch=getchar();
    }
    while(ch>='0'&&ch<='9'){
        x=x*10+ch-'0';
        ch=getchar();
    }
    return x*f;
}
inline void print(__int128 x){
    if(x<0){
        putchar('-');
        x=-x;
    }
    if(x>9)
        print(x/10);
```

```
    putchar(x%10+'0');
}
```

位运算

- (a ^ b) ^ (b ^ c) = c
- a & b + a ^ b = a | b
- a & b + a | b = a + b
- a + b = (a ⊕ b) + 2(a ∧ b)
- (a + b) − (a ⊕ b) = 2(a ∧ b)

匿名函数

1. 如果捕获列表为[&]，则表示所有的外部变量都按引用传递给lambda使用, &c 只会将 c 按照引用传进去
2. 如果捕获列表为[=]，则表示所有的外部变量都按值传递给lambda使用, c只会将 c 按值传进去
3. 匿名函数构建的时候对于按值传递的捕获列表，会立即将当前可以取到的值拷贝一份作为常数，然后将该常数作为参数传递，即：

```
auto dfs = [&](auto self, int x, int p) -> void
{
       for (auto y : adj[x])
       {
           if (y != p)
           {
               self(self, y, x);
               siz[x] += siz[y];
           }
       }
       if (siz[x] == 0)
       {
           siz[x] = 1;
       }
};
dfs(dfs, 0, -1);
```

格式	解释
[]	空捕获列表，Lambda不能使用所在函数中的变量。
[names]	names是一个逗号分隔的名字列表，这些名字都是Lambda所在函数的局部变量。默认情况下，这些变量会被拷贝，然后按值传递，名字前面如果使用了&，则按引用传递
[&]	隐式捕获列表，Lambda体内使用的局部变量都按引用方式传递
[=]	隐式捕获列表，Lanbda体内使用的局部变量都按值传递
[&,identifier_list]	identifier_list是一个逗号分隔的列表，包含0个或多个来自所在函数的变量，这些变量采用值捕获的方式，其他变量则被隐式捕获，采用引用方式传递，identifier_list中的名字前面不能使用&。
[=,identifier_list]	identifier_list中的变量采用引用方式捕获，而被隐式捕获的变量都采用按值传递的方式捕获。identifier_list中的名字不能包含this，且这些名字面前必须使用&。

基础课类算法

排序

快排

```
void quick_sort(int q[], int l, int r)
{
    if(l>=r) return;
    int x = q[l+r>>1] , i = l-1 , j = r+1;//先移动，这样保证一定是两段
    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

> 主要用于求逆序对和正序对的问题

```
int n;
int q[N],tmp[N];//tmp存储中间排序的结果
ll merge_sort(int l,int r)
{
    if(l>=r) return 0;
    int mid = l + r >> 1;
    ll res = merge_sort(l,mid) + merge_sort(mid+1,r);
    //归并的过程
    int k =0,i=l,j=mid+1;
    while(i <=mid && j <= r)
        if(q[i] <= q[j]) tmp[k++] = q[i++];
        else
        {
            tmp[k++] = q[j++];
            res += mid - i + 1;//一左一右
        }
    while(i<=mid) tmp[k++] = q[i++];//扫尾
    while(j<=r) tmp[k++] = q[j++];
    for(int i=l,j=0;i<=r;i++,j++)//物归原主 i循环原数组 j循环临时数组
        q[i] = tmp[j];
    return res;
}
int main()
{
    cin >> n;
    for(int i=0;i<n;i++)
        cin >> q[i];
    cout << merge_sort(0,n-1) << endl;
    return 0;
}
```

二分

```
// 区间[l, r]被划分成[l, mid]和[mid + 1, r]时使用：
int bsearch_1(int l, int r)//求满足要求的最小值,求左端点
{
    while (l < r)
    {
        int mid = l + r >> 1;
        if (check(mid)) r = mid;    // check()判断mid是否满足性质
        else l = mid + 1;
    }
    return r;
}
// 区间[l, r]被划分成[l, mid - 1]和[mid, r]时使用：
int bsearch_2(int l, int r)//求满足要求的最大值,求右端点
{
    while (l < r)
    {
        int mid = l + r + 1 >> 1;
        if (check(mid)) l = mid;
        else r = mid - 1;
    }
    return r;
}//浮点数二分注意精度即可,比要求多2位小数即可
```

高精度

```
//高精度计算时,存数据的时候都是逆序的,从后往前存,对于vector数组从后往前遍历是原数据
for(int i = a.size()-1;i>=0;i--) A.push_back(a[i] - '0');//A = [6,5,4,3,2,1]
for(int i = b.size()-1;i>=0;i--) B.push_back(b[i] - '0');
//输出的时候也要注意从后往前输出
for(int i = C.size()-1;i>=0;i--) cout << C[i];

//加法
vector<int> add(vector<int> &A,vector<int> &B)//添加引用可以节省时间
{

    vector<int> C;
    int t = 0;//进度
    for(int i=0;i<A.size() || i < B.size();i++){
        if(i < A.size()) t += A[i];
        if(i < B.size()) t += B[i];
        C.push_back(t % 10);
        t /= 10;
    }
    if(t) C.push_back(t);
    return C;
}

//减法
//判断AB之间的大小关系
bool cmp(vector<int> &A,vector<int> &B)
{
    if(A.size() != B.size()) return A.size() > B.size();//0 或 1
    for(int i = A.size() - 1;i>=0;i--)
        if(A[i] != B[i]) return A[i] > B[i]; // 0 或 1
    return true;//两者相等
}
vector<int> sub(vector<int> &A,vector<int> &B)//添加引用可以节省时间
{
    vector<int> C;
    for(int i =0,t=0; i<A.size();i++)
    {
        t = A[i] - t;
        if(i < B.size()) t -= B[i];
        C.push_back((t + 10)%10);//中和了借1和单纯减
        if(t < 0) t = 1;
        else t = 0;
    }
    while(C.size() > 1 && C.back() == 0 ) C.pop_back();//删除前导0
    return C;
}

//乘法
vector<int> mul(vector<int> &A,int b)
{
    vector<int> C;
    int t=0;  //进位
    for(int i=0;i<A.size() || t;i++) //加上t是因为最后一位乘法时可能出现进位
    {
        if(i < A.size()) t += A[i] * b;
        C.push_back(t%10);
        t /= 10;  //进位
    }
    return C;
}

//除法
vector<int> div(vector<int> &A,int b,int &r)//r必须传引用或指针因为需要其改变
{
    vector<int> C;//商
    r = 0;
    for(int i = A.size() - 1;i>=0;i--)
    {
        r = r * 10 + A[i];  //余数放在下一位相当于下一位变成10*r + A[i]
        C.push_back(r / b);
        r %= b;
    }
    reverse(C.begin(),C.end()); //因为i从大到小所以结果是倒序的反转回来
    while(C.size() > 1 && C.back() ==0) C.pop_back();
```

```
    return C;
}
```

前缀和与差分数组

```
for(int i=1;i<=n;i++)//求前缀和
        for(int j=1;j<=m;j++)
            s[j][j] = s[i-1][j] + s[i][j-1] - s[i-1][j-1] + a[i][j];
while(q--)//算子矩阵的和
    {
        int x1,y1,x2,y2;
        cin >> x1 >> y1 >> x2 >> y2;
        cout << s[x2][y2] - s[x1-1][y2] - s[x2][y1-1] + s[x1-1][y1-1];
    }
// 差分
void insert(int x1,int y1,int x2,int y2,int c)
{
    b[x1][y1] += c;
    b[x2+1][y1] -= c;
    b[x1][y2+1] -=c;
    b[x2+1][y2+1] +=c;
}//差分数组求前缀和得到当前元素
```

双指针

```
for(int i=0,j=0;i<n;i++){
        s[a[i]]++;
        while(j < i && s[a[i]] > 1)
        {
            s[a[j]]--;//j到i之间一定有一个数多次出现。
            j++;
        }
        res = max(res,i-j+1);
    }
```

lowbit

```
int lowbit(int x)//求最后一个 1所代表的值
{
    return x & -x;
}
int func(int x) //判断一个数的二进制表示中有多少 1
{
    int countx = 0;
    while(x)
    {
        countx++;
        x = x&(x-1); //清除掉整数a二进制中最右边的1。
        //x -= lowbit(x);
    }
    return countx;
}
```

离散化

保序方式

```
vector<int> alls; // 存储所有待离散化的值,这里是将下标离散化
sort(alls.begin(), alls.end()); // 将所有值排序
alls.erase(unique(alls.begin(), alls.end()), alls.end());   // 去掉重复元素

// 二分求出x对应的离散化的值
int find(int x) // 找到第一个大于等于x的位置,传入的值是下标,二分查找vector里面等于这个数存值,查找左端点和右端点求区间和
{
    int l = 0, r = alls.size() - 1;
    while (l < r)
    {
        int mid = l + r >> 1;
        if (alls[mid] >= x) r = mid;
        else l = mid + 1;
    }
```

```
    return r + 1; // 映射到1, 2, ...n,+1是离散化下标从1开始，不加从0开始
}
```

无序map

```
unordered_map<int,int> mp;//增删改查的时间复杂度是 O(1)
int res;
int find(int x)
{
    if(mp.count(x)==0) return mp[x]=++res;
    return mp[x];
}
```

RMQ(ST表查询区间最值)

```
//以查询最大值为例,ST表查询主要是2倍的跳
状态表示：  集合：f(i,j)表示从位置i开始长度为2^j的区间的最大值；
            属性：MAX
状态转移：  f(i,j)=max(f(i,j-1),f(i+(1<<(j-1)),j-1));
            含义：把区间[i,i+2^j],分成两半，[i,i+2^(j-1)]和[i+(1<<(j-1)),2^j],整个区间最大值就是这两段区间最大值的最大值
const int N=2e5+7,M=20;
int dp[N][M]; //存储区间最大值
int a[N];//存放每个点的值
//dp求从位置i开始长度为2^j的区间的最大值
for(int j=0;j<M;j++)
{
    for(int i=1;i+(1<<j)-1<=n;i++)
    {
        if(!j) dp[i][j]=a[i];
        else dp[i][j]=max(dp[i][j-1],dp[i+(1<<(j-1))][j-1]);
    }
}
//求任意区间的最大值；（可以预处理log）
int res=log(b-a+1)/log(2);
cout <<max(dp[a][res],dp[b-(1<<res)+1][res])<<endl;
}
```

数据结构

链表

单链表

```
int h = -1,idx = 0;
//将x插入下标是k的点的后面
void add(int k,int x)
{
    e[idx] = x,ne[idx] = ne[k],ne[k] = idx,idx++;
}
//将下标是k的点后面的点删除
void remove(int k)
{
    ne[k] = ne[ne[k]];//只需要更新指向即可，跳过下一个点，指向下下个点
}
```

双链表

```
//在第k个点的右边插入一个点x---在k的左边插入时传入 l[k],x 即可
void add(int k,int x)
{
    e[idx] = x;
    r[idx] = r[k];
    l[idx] = k;
    l[r[k]] = idx;//左右更新不能写反，右节点的左指针更新需要用到左节点的右指针
    r[k] = idx;

}
//删除第 k 个点
void remove(int k)
{
```

```
    r[l[k]] = r[k];
    l[r[k]] = l[k];
}
```

Tire树

```
int son[N][26],cnt[N],idx=0;//下标是0的点，既是根节点，又是空节点 idx存的是当前用到的节点
char str[N]; //son数组模拟的指针-下标是x的点，x节点的儿子存在son里面，cnt存的是以x结尾的单词数量
void insert(char str[])
{
    int p = 0;
    for(int i=0;str[i];i++)
    {
        int u = str[i] - 'a';//将字母映射到0~26的数字表示
        if(!son[p][u]) son[p][u] = ++idx;//p节点不存在u这个字母-新建一个节点
        p = son[p][u];//走到下一个节点
    }//次数的!son[p][u]决定了是++idx不是idx++，因为son不应该等于0
    cnt[p]++;//表示以对应的u结尾的单词个数加 1
}
int query(char str[])//查询操作是返回单词出现多少次
{
    int p=0;
    for(int i=0;str[i];i++)
    {
        int u = str[i] - 'a';//将字母转化为对应的子节点的编号
        if(!son[p][u]) return 0; //不存在，表示没有这个单词
        p = son[p][u];
    }

    return cnt[p];//返回单词个数
}
```

并查集

> - 并查集维护集合内有多少点

```
for(int i=1;i<=n;i++) p[i] = i,si[i]=1;//将所有节点的父节点先赋值为自己
if(find(a) == find(b)) continue;//a和b在一个集合里
si[find(b)] += si[find(a)];//数据个数相加
p[find(a)] = find(b);   //si数组存储的即是并查集内的所有点的数量
```

带权并查集

```
int dis[100001];//记录该集合的总大小
int tot[100001];//前面的节点数
int fa[100001];//记录元素祖先
void clean(int n)//初始化
{
    for (int i=0;i<=n;i++)
        fa[i]=i,tot[i]=1;
}
int find(int k)//查找该元素的祖先
{
    if(fa[k] == k)
        return k;
    int f = find(fa[k]);
    dis[k] += dis[fa[k]];
    return fa[k];
}
void add(int x,int y)//合并两个集合
{
    x = find(x);
    y = find(y);
    dis[x] = tot[y];//a 树作为 b 树的子树
    tot[y] += tot[x];
    fa[x] = y;
}
```

扩展域并查集

> - 拓展域并查集解决了一种多个有相互关系的并查集，放在一起考虑的问题,拓展域并查集讲的是多个集合，之间有相互关系一般为相互排斥关系，判断是否在一个集合等。

```
典型例题 ---- 食物链
//判断到根节点的距离即可 距离%3==0 与根节点是同一类 %3==1 可以吃根节点 %3==2,被根节点吃 ，路径压缩的时候维护距离即可
int n,m;//动物数与说话个数
int p[N],d[N];//并查集和维护的距离
int find(int x)
{
    if(p[x] != x)//x不是根
    {
        int t = find(p[x]);//不能直接写p[x] = find(p[x])
        d[x] += d[p[x]];//直接写的话，p[x]会改变，不是初始x的p
        p[x] = t;
    }
    return p[x];
}
int main()
{
    cin >> n >> m;
    for(int i=1;i<=n;i++) p[i] = i;//距离不必初始化全局变量为0，表示其实都是与自己同一类
    int res = 0;
    while(m --)
    {
        int t,x,y;
        cin >> t >> x >> y;
        if(x > n || y > n)
            res ++;
        else
        {
            int px = find(x),py = find(y);
            if(t==1)
            {    //父节点相同是前提 然后不是都%3=0，不是同一类
                if(px == py && (d[x] - d[y])%3 != 0) res ++;
                else if(px != py)
                {
                    p[px] = y;
                    d[px] = d[y] - d[x];
                }
            }
            else
            {//X吃Y   X到根的距离应该比Y到根多 1 （mod 3）的情况下
                if(px == py && (d[x] - d[y] - 1 ) %3 != 0) res ++;
                else if(px != py)
                {
                    p[px] = py;//先合并
                    d[px] = d[y] + 1 - d[x];
                }
            }
        }
    }
    cout << res << endl;
    return 0;
}
```

搜索

DFS

- 深度搜索,DFS最显著的特征在于其 **递归调用自身** ,同时与BFS类似,DFS会对其访问过的点打上访问标记,在遍历图时跳过已打过标记的点,以确保**每个点仅访问一次**,在 DFS 过程中，通过记录每个节点从哪个点访问而来，可以建立一个树结构，称为 DFS 树。DFS 树是原图的一个生成树。

```
int n;
char g[N][N];//路径
bool col[N],dg[N],udg[N];
//col表示该列是否有皇后 dg表示对角线 udg表示反对角线
/*第一种搜索顺序*/
void dfs(int u)
{
    if(u == n)//行
    {
```

```
        for(int i=0;i<n;i++)
            puts(g[i]);
        puts("");
        return;
    }
    for(int i=0;i<n;i++)
        if(!col[i] && !dg[u+i] && !udg[n - u + i])
        {
            g[u][i] = 'Q';
            col[i] = dg[u + i] = udg[n - u + i] = true;
            dfs(u + 1);
            col[i] = dg[u + i] = udg[n - u + i] = false;
            g[u][i] = '.';
        }
}
int main01()
{
    cin >> n;
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
            g[i][j] = '.';
    dfs(0);
    return 0;
}
```

BFS

- 广度搜索,BFS算法找到的路径是从起点开始的 **最短合法路径**，一般借助队列存储要处理的点,bool 数组记录是否访问过, 在 BFS 过程中，通过记录每个节点从哪个点访问而来，可以建立一个树结构，即为 **BFS 树**.

```
#include <iostream>
#include<algorithm>
#include<cstring>
using namespace std;
typedef  pair<int,int> PII;
const int N = 110;
int n,m;
int g[N][N];//g数组用来存储输入的图
int d[N][N];//d数组用来每一个点到起点的距离
PII q[N*N],Prev[N][N];//手写队列，可以直接用queue,prev记录这个路是从哪一点过来的
int bfs()
{
    int hh=0,tt=0;//头 尾 - 用来表示 点的移动
    q[0] = {0,0};
    memset(d,-1,sizeof(d));//距离初始化，表示没有走过
    d[0][0] = 0;
    int dx[4] = {-1,0,1,0},dy[4]={0,1,0,-1}; // 表示一个点向四个方向找下一层是，坐标的变化
    while(hh <= tt)//队列不空
    {
        auto t = q[hh++];//每一次取出队头元素
        for(int i=0;i<4;i++)//上下左右四个方向判断
        {//用向量表示移动情况
            int x = t.first + dx[i],y=t.second + dy[i];//x y表示沿着这个方向走可以走到的点
            if(x>=0 && x<n && y>=0 && y<m && g[x][y] == 0 && d[x][y]==-1)
            { // 在图的范围内                    是通路 - 0       没有走过
                d[x][y] = d[t.first][t.second] + 1; //更新这个点的距离
                Prev[x][y] = t;//这个队列从t点过来的，即队列的头，终点未存进队列中
                q[++tt] = {x,y};//把这个点存进队列中                 当hh = n时跳出循环了
            }//Prev队列中存储的最后一个点是终点的前一个点，终点距离更新了，但终点未存进队列
        }
    }
    int x = n-1,y = m-1;//表示路径，看题目是否要求 - 此处逆序的
    while(x || y)
    {
        cout << x <<' ' << y << endl;
        auto t = Prev[x][y];//最后一点未存进队列中
        x = t.first,y= t.second;//死记-先存y再存x
    }
    return d[n-1][m-1];//输出右下交的点代表的距离 - 看出口在哪里
}
int main()
{
    cin >> n >> m;
    for(int i=0;i<n;i++)
        for(int j=0;j<m;j++)
```

```
            cin >> g[i][j];

    cout << bfs() << endl;
    return 0;
}
```

双端BFS

- 又称 0-1 BFS,即是边权为 0 或 1的最短路,**实现：**一般情况下把没有边权的边扩展到的点放到队首,有权值的边扩展到的点放到队尾,.这样即可 保证像普通 BFS一样整个队列队首到队尾权值单调不下降　cf 173B

```
#define INF (1 << 29)
int n, m;
char grid[1001][1001];
int dist[1001][1001][4];
int fx[] = {1, -1, 0, 0};
int fy[] = {0, 0, 1, -1};
deque<int> q;  // 双端队列
void add_front(int x, int y, int dir, int d) {  // 向前方加
  if (d < dist[x][y][dir]) {
    dist[x][y][dir] = d;
    q.push_front(dir);
    q.push_front(y);
    q.push_front(x);
  }
}
void add_back(int x, int y, int dir, int d) {  // 向后方加
  if (d < dist[x][y][dir]) {
    dist[x][y][dir] = d;
    q.push_back(x);
    q.push_back(y);
    q.push_back(dir);
  }
}
int main() {
  cin >> n >> m;
  for (int i = 0; i < n; i++) cin >> grid[i];

  for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
      for (int k = 0; k < 4; k++) dist[i][j][k] = INF;
  add_front(n - 1, m - 1, 3, 0);
  while (!q.empty()) {  // 具体搜索的过程，可以参考上面写的题解
    int x = q[0], y = q[1], dir = q[2];
    q.pop_front();
    q.pop_front();
    q.pop_front();
    int d = dist[x][y][dir];
    int nx = x + fx[dir], ny = y + fy[dir];
    if (nx >= 0 && nx < n && ny >= 0 && ny < m)
      add_front(nx, ny, dir, d);  // 判断条件
    if (grid[x][y] == '#')
      for (int i = 0; i < 4; i++)
        if (i != dir) add_back(x, y, i, d + 1);
  }
  if (dist[0][0][3] == INF)
    cout << -1 << endl;
  else
    cout << dist[0][0][3] << endl;
  return 0;
}
```

ST表　ST

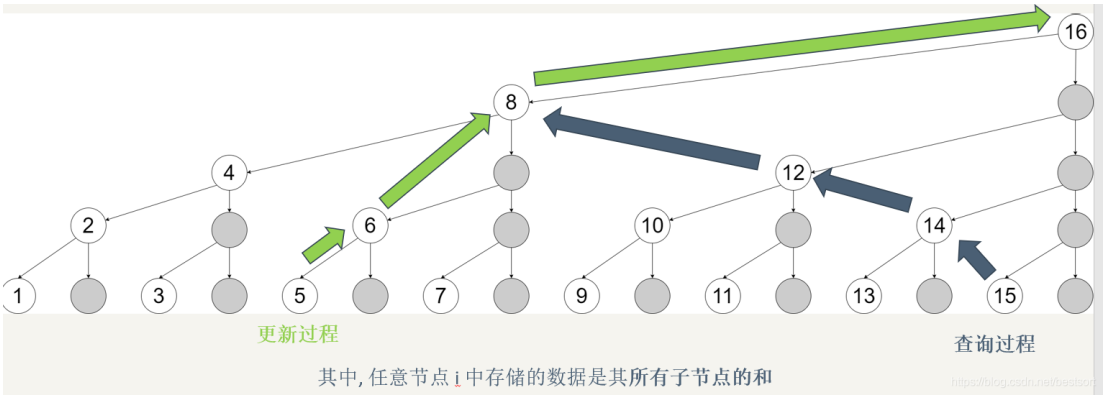
- ST表是用于解决 **可重复贡献问题** 的数据结构.可贡献问题指的是对于运算 opt,满足 x opt x = x,则对应的区间查询就是一个可重复贡献问题.例如 RMQ(区间最值),和max,gcd 类,但是区间和这种区间一旦重复区间和就一定会改变的不是,同时**opt必须满足结合律才能使用ST表**.O(nlogn) 询问,O(1)查询,但不支持区间修改,**二维DP表示从 i 长度为**
  2^j 的区间内的最值, 如此可以直接预处理所有长度为 2^j 的区间内的最值, 询问的时候将长度分为两个前后区间查询

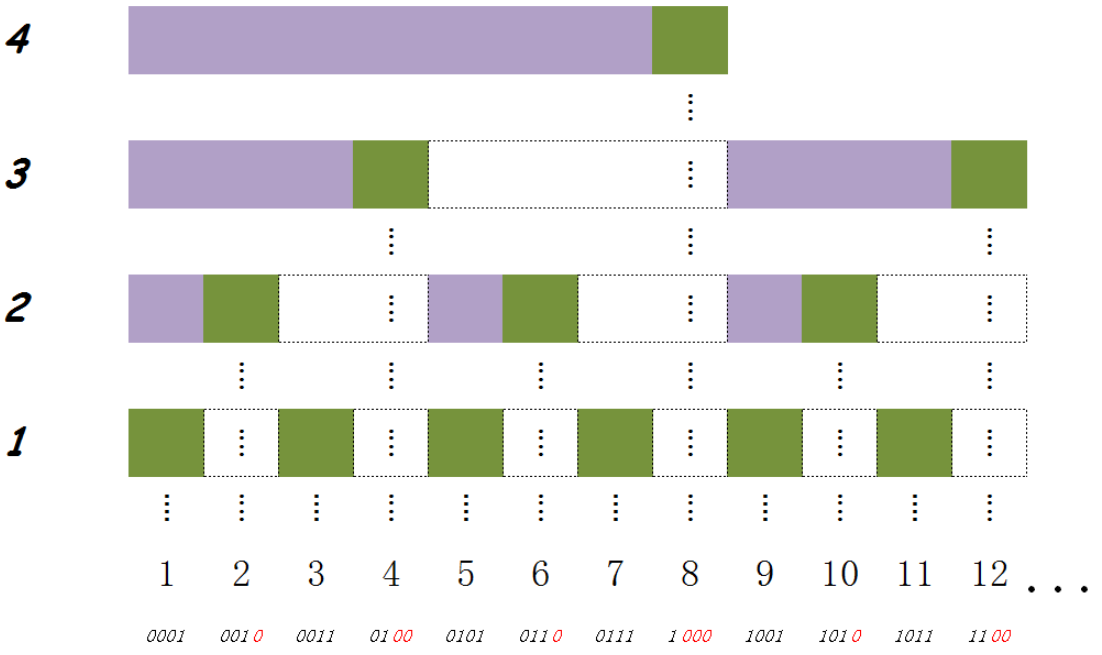
```
const int N = 1e5 + 10;//数据比较极限,需要加快读,否则超时.     可以证明 2 * pow(2,__lg(len)) >= len
int f[N][22];//长度为2^j次方,从i开始的最大值
int n, m;
int Log[N * 2];//因为log(a)表示小于等于a的2的最大几次方。即 2 ^ log <= a,即是log(a)下取整
inline int read()
```

```
{
    int x=0,f=1;char ch=getchar();
    while (ch<'0'||ch>'9'){if (ch=='-') f=-1;ch=getchar();}
    while (ch>='0'&&ch<='9'){x=x*10+ch-48;ch=getchar();}
    return x*f;
}
int main()
{
    n = read(),m = read();
    for (int i = 1; i <= n; i++)
        cin >> f[i][0];//从i开始长度为 1 的最大值即为自己
    //更新f数组  转换为长度累加即可
    for (int i = 1; i < 22; i++)
    {
        for (int j = 1; j + (1 << i) - 1 <= N; j++)
        {//拆分为左右两等段
            f[j][i] = max(f[j][i - 1], f[j + (1 << (i - 1))][i - 1]);
        }
    }
    int l ,r ;
    for (int i = 0; i < m; i++)
    {
        l = read(),r = read();
        int s = __lg(r - l + 1);//__lg自带的log2的函数,返回的是log(n),下取整
        cout << max(f[l][s], f[r - (1 << s) + 1][s]) << endl;//为了防止越界或者不足,右边的区间起点用r-len
    }//所以l到r的最小值可以表示为min(从l往后2^t的最小值，从r往前2^t的最小值)
    return 0;
}

```

树状数组





单点修改求区间和

> - lowbit函数,add函数,sum函数都是基于二进制,**树状数组存的是区间前缀和** 洛谷 楼兰图腾 ,

```cpp
const int N = 2e5 + 10;//树状数组维护的是值不是坐标
int a[N];//原数组
int tr[N];//表示树状数组,表示y=i的高度出现了几次,树状数组tr本身存的就是前缀和,基于二进制跳动前缀和,不是常规前缀和
int great[N],lower[N];
int n;
int lowbit(int x)
{
    return x & -x;
}//返回最后一个1及后面的0
void add(int x,int c)//在x的位置加上c
{
    for(int i=x;i<=n;i+=lowbit(i))//基于二进制前缀和,除了第一个读入的数据,存到 10,100,110,1000这种整二进制类型
        tr[i] += c;
}
int sum(int x)//查询操作
{
    int res = 0;
    for(int i=x;i;i-=lowbit(i))//前缀和,tr的子节点的求法,计算常规前缀和.
        res += tr[i];
    return res;
}
int main()//树状数组中存的是高度出现次数的前缀和,不是区间下标
{
    _;
    cin >> n;
    for(int i=1;i<=n;i++)//对应的y
        cin >> a[i];
    for(int i=1;i<=n;i++)//先从左向右算一遍
    {
        int y = a[i];//下为前缀和求区间和,因为是全排列,所以最大为 n, y - 1的差值,为区间和
        great[i] = sum(n) - sum(y);//高度大于等于y + 1里面有多少数
        lower[i] = sum(y - 1);//高度为 1 到 y - 1 (包含)里面有多少数
        add(y,1);
    }
    memset(tr,0,sizeof tr);
    ll res1 = 0,res2 = 0;
    for(int i=n;i;i--)//倒序求出来 V 型 和 ^ 型
    {
        int y = a[i];
        res1 += great[i] * (ll)(sum(n) - sum(y));
        res2 += lower[i] * (ll)sum(y - 1);
        add(y,1);
    }
    cout << res1 << " " << res2 << endl;
    return 0;
}
```

区间修改单点求和

- 树状数组维护的是差分,树状数组 tr 记录的是差分的前缀和,即是当前元素(基于二进制,需要转换为前缀和)

```cpp
const int N = 1e5 + 10;
int n,m,a[N];//原数组
int tr[N];//差分数组,树状数组维护的前缀和是基于lowbit,二进制的最后一位 1 所代表的大小所跳动的,不是基于 1
int lowbit(int x)
{
    return x & -x;
}
void add(int x,int c)
{
    for(int i=x;i<=n;i+=lowbit(i))
        tr[i] += c;
}
ll sum(int x)//求和
{
    ll res = 0;
    for(int i=x;i;i-=lowbit(x))
        res += tr[i];
    return res;
}
int main()
```

```
{
    _;
    cin >> n >> m ;
    for(int i=1;i<=n;i++)
       cin >> a[i];//先读入原数组
    for(int i=1;i<=n;i++)
        add(i,a[i] - a[i -1]);//转换为差分数组
    while(m --)
    {
        char op[2]  ;
        int l,r ,d;
        cin >> op >> l;
        if(op[0] == 'C')
        {//修改区间,利用差分的性质,修改 l 和 r+1 两个位置即可,tr 求的二进制下的前缀和
            cin >> r >> d;
            add(l,d),add(r+1,-d);
        }
        else
            cout << sum(l) << endl;//需要将二进制转换为常规前缀和
    }
    return 0;
}
```

区间修改区间求和

- 区间修改加上或减去**同一个值**,树状数组维护的是数组b[]的前缀和,b是读入数组的**差分数组**,tr存的是差分数组的二进制前缀和,建立一个(n+1)*(n+1)的每一行
  为a_1到 a_n 的矩阵发现,区间和等于,$\sum_1^n a_i * (n+1) - \sum_1^n i * a_i$,所以建立两个tr数组通过树状数组求解

```
ll tr1[N];//树状数组维护b[]的前缀和(b是a的差分数组)
ll tr2[N];//维护b[i] * i 的前缀和
int n,m;//数组长度和操作个数
int lowbit(int x)
{
    return x & -x;
}
void add(ll tr[],int x,ll c)
{
    for(int i=x;i<=n;i+=lowbit(i))
        tr[i] += c;
}
ll sum(ll tr[],int x)
{
    ll res = 0;
    for(int i=x;i;i-=lowbit(i))
        res += tr[i];
    return res;
}
ll prefix_sum(int x)
{
    return sum(tr1,x) * (x + 1) - sum(tr2,x);
}
int main()
{
    _;
    cin >> n >> m;
    for(int i=1;i<=n;i++)
        cin >> a[i];
    for(int i=1;i<=n;i++)
    {
        int b = a[i] - a[i - 1];
        add(tr1,i,b);
        add(tr2,i,(ll)b * i);
    }
    while(m --)
    {
        char op[2];
        int l,r,d;
        cin >> op >> l >> r;
        if(op[0] == 'Q')
            cout << prefix_sum(r) - prefix_sum(l-1) << endl;
        else
        {
            cin >> d;
            add(tr1,l,d),add(tr2,l,l * d);//修改a[l] += d;
```

```
            add(tr1,r+1,-d),add(tr2,r+1,-(r+1)*d);//a[r+1] -= d;
        }
    }
}
```

- 区间中每个位置添加或减去的是**差分数组**,维护的差分数组的差分即是**二维差分**,建立树状数组的方法发现,即给 l 位置+ k , l + 1 位置 + (d−k) , 给r + 1 位置 −(k+len∗d), 给 r+2 位置 +(k+(len−1)∗d).

```
void add(int x, int c) {
    for (int i = x; i <= n; i += lowbit(i)) {
        t1[i] += c, t2[i] += 1ll * x * c;
    }
}
ll ask(int x) {
    ll res = 0;
    for (int i = x; i; i -= lowbit(i)) res += (x + 1) * t1[i] - t2[i];
    return res;
}
add(i, x), add(i + 1, -x);//读入数据是同理同样需要四个位置
add(i + 1, -x), add(i + 2, x);
len = r - l + 1;//处理改变的差分数组
add(l, k), add(l + 1, d - k);
add(r + 1, -k - len * d), add(r + 2, k + (len - 1) * d);
```

- 树状数组结合二分求解,给定每一个数前面有多少比自己小的数,求每个数是多少,这些数是排列,从后往前遍历,确定一个数后add(i,-1)将这个数减去

```
 for(int i=1;i<=n;i++)
     tr[i] = lowbit(i);//每一个数都是 1,tr前缀和就是lowbit(i)
// for(int i=1;i=n;i++)
//     add(i,1);//两种初始化操作
for(int i=n;i;i--)
{//h[i]是输入的数,表示前面有多少比它小的数
    int k = h[i] + 1;//算上自己前面一共 k 头牛
    int l = 1,r = n;//二分
    while(l < r)//查找第一个大于等于k的数,是排列,第一个等于k的数
    {
        int mid = l + r >> 1;
        if(sum(mid) >= k) r = mid;
        else l = mid + 1;
    }//int r = upper_bound(sum + 1,sum + 1 + n,k) - sum;
    ans[i] = r;
    add(r,-1);
}
```

二维树状数组

单点修改,子矩阵查询

二维树状数组,用来维护**二维数组**上的单点修改和前缀信息.,与一维树状数组类似,

线段树

李超线段树

区间最值操作 & 区间历史最值

划分树

二叉搜索树&平衡树

跳表

可持久数据结构

树套树

K-D Tree

动态树

析合树

PQ树

手指树

霍夫曼树

图论

树和图

> - 树的重心指的是把这个点删去后,剩余联通块中点最大值最小,满足这个条件的点是树的重心.

```cpp
const int N = 100010,M = N*2;//删除树中的一个点，使得剩下的连通块中点数最大值的最小值
int h[N],e[M],ne[M],idx=0;//h存的是n个链表的链表头，e  ne  idx与链表一致
bool st[N];//idx存的是边
int ans = N;//最小的最大值
void add(int a,int b)//在a后插入b
{
    e[idx] = b,ne[idx] = h[a],h[a] = idx++;
}
int dfs(int u)//以u为根的子树中点的数量
{
    st[u] = true;//标记一下，这点点已经被搜过了
    int sum = 1,res = 0;//res是每一个连通块的最大值
    for(int i=h[u];i!=-1;i=ne[i])
    {
        int j = e[i];
        if(!st[j])
        {
            int s = dfs(j);//可能会分为多个连通块
            res = max(res,s);//把这个点删除后每一个联通块点的最大值
            sum += s;//这个连通块一共多少点算上当前u
        }  //相减的剩下的点数
    } //n - sum 代表这个指向这个点的连通块点数
    res = max(res,n-sum);//被删除的连通块和剩下的连通块
    ans = min(ans,res);
    return sum;
}
int main()
{
    cin >> n;
    memset(h,-1,sizeof h);//初始化，将头全指向-1，代表结尾
    for(int i=0;i<n-1;i++)
    {
        int a,b;
        cin >> a >>b;
        add(a,b),add(b,a);//无向边
    }
    dfs(1);//从第一个节点开始深搜-搜的是图的节点编号
    cout << ans << endl;
}
```

树上问题

树基础

- 适用于有根树和无根树

> **森林**，**生成树：** 一个连通无向图的生成子树,同时要求是树.也即在图的边集中选择 n - 1 条,将所有顶点连通, **无根图的叶节点：** 度数不超过 1 的结点,不超过 1 是因为 n = 1时的特例 ,**有根图的叶结点：** 没有子结点的结点.

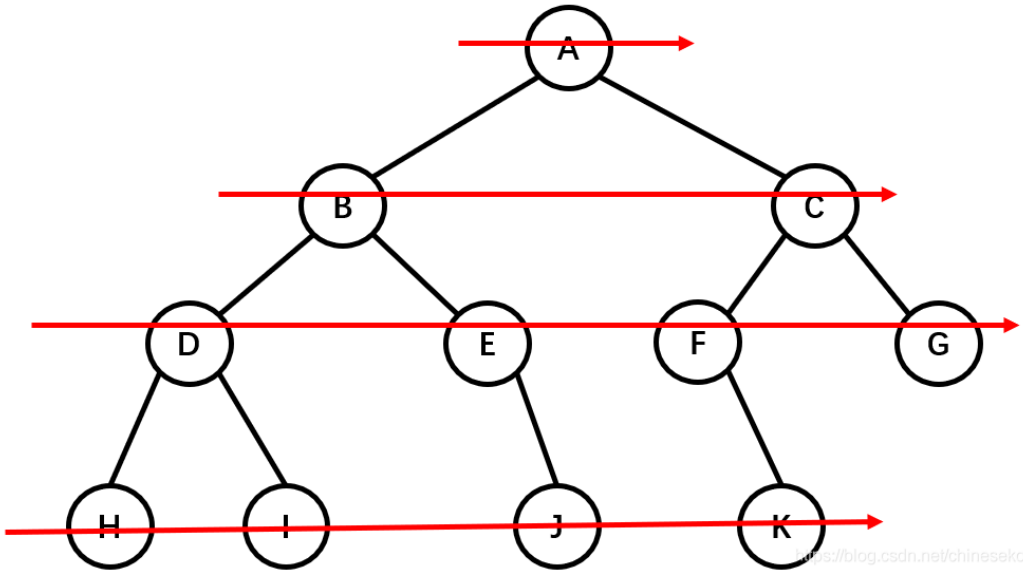
- 只适用于有根树

> **父节点,子节点,祖先,兄弟,后代,, 结点的深度：** 到根节点的路径上的边数 ; **树的高度：** 所有结点的深度的最大值 ; **子树：** 删掉与父亲相连的边后,该结点所在的子图 ;

- 特殊的树

> **链：**满足与任一结点相连的边不超过 条的树称为链。**菊花图：**满足存在 使得所有除 以外结点均与 相连的树称为菊花。**有根二叉树：**每个结点最多只有两个儿子（子结点）的有根树称为二叉树。常常对两个子结点的顺序加以区分，分别称之为左子结点和右子结点。大多数情况下，**二叉树** 一词均指**有根二叉树**。**完整二叉树：** 每个结点的子结点数量均为 0 或者 2 的二叉树。换言之，每个结点或者是树叶，或者左右子树均非空。**完全二叉树：**只有最下面两层结点的度数可以小于 2，且最下面一层的结点都集中在该层最左边的连续位置上。**完美二叉树：**所有叶结点的深度均相同，且所有非叶节点的子节点数量均为 2 的二叉树称为完美二叉树。

树的遍历 - 主要先 中 后三种



树上DFS

> 在树上DFS指的是 : 先遍历根节点,然后分别访问根节点每个儿子的子树,可以用来求出每个节点的深度,父亲等信息..

二叉树上DFS - 先序遍历

- 按照 **根,左,右** 的顺序遍历二叉树,即是从根节点出发,一直向下遍历左儿子直至到叶节点,然后寻找上一个父节点的右儿子,接着重复第一步遍历左儿子,**DFS**的思路一条路走到结尾在回溯. 结果是 **A B D H I E J C F K G**

二叉树上DFS - 中序遍历

- 按照 **左,根,右** 的顺序遍历二叉树,即是先找到最左边的叶节点,然后向上遍历找到最近的父节点,遍历这个父节点的右儿子,到叶节点后继续线上回溯,找这个父节点的最近父节点遍历右儿子,递归至根节点,**中序遍历可以看成 :** 二叉树每个节点,垂直投影下来(可以理解为每个节点从最左边开始垂直掉到地上),然后从左向右数,得到的结果便是中序遍历的结果,结果是 **H D I B E J A F K C G** (K 是 F 的右儿子,所以 K 在 F 后)

二叉树上DFS - 后序遍历

- 按照 **左,右,根** 的顺序遍历二叉树,就像是剪葡萄,将一串葡萄剪成一颗一颗的,每次剪掉的必须是一颗不是一串.结果是 **H I D J E B K F G C A**

层次遍历 BFS

- 从根节点开始,一层一层,从上到下,每层从左到右,依次写值就可以,结果是 **A B C D E F G H I J K**

```c
#include<stdio.h>
#include<stdlib.h>
typedef struct Tree
{
 int data;                  //  存放数据域
 struct Tree *lchild;           //  遍历左子树指针
 struct Tree *rchild;           //  遍历右子树指针
}Tree,*BitTree;
BitTree CreateLink()
{
    int data;
    int temp;
    BitTree T;
    scanf("%d",&data);      //  输入数据
    temp=getchar();         //  吸收空格

    if(data == -1)  //  输入-1 代表此节点下子树不存数据，也就是不继续递归创建
        return NULL;
```

```
    else
    {
        T = (BitTree)malloc(sizeof(Tree));          //      分配内存空间
        T->data = data;                             //      把当前输入的数据存入当前节点指针的数据域中

        printf("请输入%d的左子树: ",data);
        T->lchild = CreateLink();                   //      开始递归创建左子树
        printf("请输入%d的右子树: ",data);
        T->rchild = CreateLink();                   //      开始到上一级节点的右边递归创建左右子树
        return T;                           //      返回根节点
    }

}
void ShowXianXu(BitTree T)          //      先序遍历二叉树
{
    if(T==NULL)                     //  递归中遇到NULL，返回上一层节点
        return;
    printf("%d ",T->data);
    ShowXianXu(T->lchild);          //  递归遍历左子树
    ShowXianXu(T->rchild);          //  递归遍历右子树
}
void ShowZhongXu(BitTree T)         //      中序遍历二叉树
{
    if(T==NULL)                     //  递归中遇到NULL，返回上一层节点
        return;
    ShowZhongXu(T->lchild);         //  递归遍历左子树
    printf("%d ",T->data);
    ShowZhongXu(T->rchild);         //  递归遍历右子树

}
void ShowHouXu(BitTree T)           //      后序遍历二叉树
{
    if(T==NULL)                     //  递归中遇到NULL，返回上一层节点
        return;
    ShowHouXu(T->lchild);           //  递归遍历左子树
    ShowHouXu(T->rchild);           //  递归遍历右子树
    printf("%d ",T->data);
}
int main()
{
    BitTree S;
    printf("请输入第一个节点的数据:\n");
    S = CreateLink();           //      接受创建二叉树完成的根节点
    printf("先序遍历结果: \n");
    ShowXianXu(S);              //      先序遍历二叉树
    printf("\n中序遍历结果: \n");
    ShowZhongXu(S);             //      中序遍历二叉树
    printf("\n后序遍历结果: \n");
    ShowHouXu(S);               //      后序遍历二叉树
    return 0;
}
```

反推

- 已知中序遍历和另外一个序列可以求第三个序列.前序的第一个是 `root`，后序的最后一个是 `root`。先确定根节点，然后根据中序遍历，在根左边的为左子树，根右边的为右子树。对于每一个子树可以看成一个全新的树，仍然遵循上面的规律。

无根树

- 树的遍历一般为深度优先遍历,这个过程要注意的是避免重复访问节点,由于树是无环图,因此只需记录当前节点是由哪个节点访问而来,此后进入除该节点外的所有相邻结点,即可避免重复访问.

```
void dfs(int u, int from) {
  // 递归进入除了 from 之外的所有子结点
  // 对于出发结点，from 为空，故会访问所有相邻结点，这与期望一致
  for (int v : adj[u])
    if (v != from) {
      dfs(v, u);
    }
}
// 开始遍历时
int EMPTY_NODE = -1;  // 一个不存在的编号
int root = 0;         // 任取一个结点作为出发点
dfs(root, EMPTY_NODE);
```

树的直径

> 树上任意两节点之间最长的简单路径即为树的直径.可以使用两次 DFS 或者 树形DP 的方法在 O(n) 时间求出树的直径.树的直径是相当于点来说

- 两遍DFS : **定理：** 在一棵树上,从任意节点 y 开始进行一次 DFS,到达的距离其最远的节点 z 必为直径的一端,从节点 z 找另一端.(两次 DFS 适用于边权不为负,如果含有负权边则不能使用两次 DFS 求解).如果需要求出一条直径上的所有结点,则可以在第二次 DFS 的过程中,记录每个点的前序结点,即可从直径的一端一路向前,遍历直径上的所有结点.

```
const int N = 10000 + 10;
int n, c, d[N];
vector<int> E[N];
void dfs(int u, int fa) {
  for (int v : E[u]) {
    if (v == fa) continue;
    d[v] = d[u] + 1;
    if (d[v] > d[c]) c = v;
    dfs(v, u);
  }
}
int main() {
  cin >> n;
  for (int i = 1; i < n; i++) {
    int u, v;
    cin >> u >> v;
    E[u].push_back(v), E[v].push_back(u);
  }//使用vector的方式存树.
  dfs(1, 0);
  d[c] = 0, dfs(c, 0);
  cout << d[c] << endl;
  return 0;
}
```

- 树形DP : 记录当 1 为 树的根时,每个节点作为子树的根向下,所能延伸的最长路径
  d_1 与次长路径(与最长路径无公共边)长度为 d_2 ,那么直径就是对于每一个点,
  该点 $d_1 + d_2$ 能取到的值中的最大值,树形DP可以在存在负权边的情况下求解出树的直径.,如果需要求出一条直径上的点,则可以在 DP 的过程中,记录下每个
  节点能向下延伸的最长路径与次长路径所对应的子节点,在求d的同时记下对应的节点u,使得$d = d_1[u] + d_2[u]$,即可分别沿着从 u 开始的最长路径
  对应的子节点一路向某个方向(对于无根树,虽然这里指定了 1 为树的根,但仍需记录每点跳转的方向,对于有根树,一路向上跳即可),遍历直径上的所

```
const int N = 10000 + 10;
int n, d = 0;
int d1[N], d2[N];
vector<int> E[N];
void dfs(int u, int fa) {
  d1[u] = d2[u] = 0;
  for (int v : E[u]) {
    if (v == fa) continue;
    dfs(v, u);
    int t = d1[v] + 1;//t 存的是从 v 走到 u 表示直径要加一
    if (t > d1[u])
      d2[u] = d1[u], d1[u] = t;
    else if (t > d2[u])
      d2[u] = t;
  }
  d = max(d, d1[u] + d2[u]);
}
int main() {
  cin >> n;
  for (int i = 1; i < n; i++) {
    int u, v;
    cin >> u >> v;
    E[u].push_back(v), E[v].push_back(u);
  }
  dfs(1, 0);
  cout << d << endl;
  return 0;
}
```

> 性质 : 若树上所有边权均为正,则树的所有直径中点重合.

最近公共祖先 - LCA

- 两个点的最近公共祖先,就是这两个点的公共祖先里面,离根最远的点.

> 性质：**1.** LCA({u}) = u，**2.** u 是 v 的祖先,当且仅当LCA({u,v}) = u，**3.**如果 u 不为 v 的祖先并且 v 不为 u 的祖先,那么 u, v 分别处于 LCA(u,v) 的两棵不同的子树中，**4.**前序遍历中,LCA(S) 出现在所有 S 中元素之前,后续遍历中 LCA(S) 则出现在所有 S 中元素之后，**5.** 两点集并的最近公共祖先为两点集分别的最近公共祖先的最近公共祖先,即是$LCA(A \cup B) = LCA(LCA(A), LCA(B))$，**6.**两点的最近公共祖先必定处在树上两点间的最短路上，**7.**d(u,v) = h(u) + h(v) - 2h(LCA(u,v)), 其中 d 是树上两点点的距离, h 代表某点到树根的距离.

- 动态规划加倍增优化：

```cpp
int e[N],ne[N],h[500010],idx;
void add(int a,int b)
{
    e[idx] = b,ne[idx] = h[a],h[a] = idx ++;
}
int dp[500010][20];//用来保存父节点,表示第 2^j 个祖先是 x
int depth[500010]; //保存深度
bool st[500010];//预处理出父亲切点
void DFS(int k)
{
    //预处理出DP数组
    for(int i=1;(1<<i)<depth[k];i++)
        dp[k][i] = dp[dp[k][i-1]][i-1];
    for(int i=h[k];i;i=ne[i])
    {
        int j = e[i];
        //求解直接公共祖先:
        if(st[j]) continue;
        st[j] = true;
        depth[j] = depth[k]+1;
        dp[j][0] = k;// 2 ^ 0 表示第一个祖先
        DFS(j);
    }
    return;
}
int lca(int u,int v)
{
    if(depth[u] < depth[v]) swap(u,v);
    //弹节点
    int k=log2(depth[u]-depth[v]);//返回的浮点数由int接受等价于__lg,二进制最后一个 1 是第几位
    for(int i=k;i>=0;i--)
        if(depth[dp[u][i]]>=depth[v])u=dp[u][i];
    if(u == v)
        return u;
    //查询
    k = log2(depth[u]);//返回的是浮点数,但是int接受直接下取整
    for(int i=k;i>=0;i--)
    {
        if(dp[u][i] == dp[v][i])
            continue;
        u = dp[u][i];
        v = dp[v][i];
    }
    return dp[u][0];
}
int main()
{
    //LCA模板题
    int n,m,s;
    cin >> n >> m >> s;
    int u,v;
    for(int i=0;i<n-1;i++)
    {
        cin >> u >> v;
        add(u,v),add(v,u);
    }
    //这里要初始化为1，避免与深度为0的0产生歧义。
    depth[s] = 1;
    st[s] = true;
    DFS(s);
    for(int i=0;i<m;i++)
    {
        cin >> u >> v;
        cout << lca(u,v) << endl;
```

```
    }
    return 0;
}
```

- ST表优化

```
int e[2*N],ne[2*N],h[N],idx;//欧拉序列
void add(int a,int b)
{
    e[idx] = b,ne[idx] = h[a],h[a] = idx ++;
}
int sq[2*N];
int d[N];//深度

bool st[N];
int has[N];
void DFS(int k)
{
    for(int i=h[k];i;i=ne[i])
    {
        if(st[e[i]])
            continue;
        st[e[i]] = true;
        d[e[i]] = d[k]+1;
        DFS(e[i]);
        sq[++idx] = e[i];
        has[e[i]] = idx;
        sq[++idx] = k;
    }
    return;
}
int dp[2*N][20];
int lca(int l,int r)
{
    if(l>r)swap(l,r);
    int k=log2(r-l+1);
    if(d[dp[l][k]]<d[dp[r-(1<<k)+1][k]])
        return dp[l][k];
    else
        return dp[r-(1<<k)+1][k];
}
int main()
{
    int n,m,s;
    cin >> n >> m >> s;
    int u,v;
    for(int i=0;i<n-1;i++)
    {
        cin >> u >>v;
        add(u,v),add(v,u);
    }
    idx=0;
    st[s]=true;
    d[s]=1;
    DFS(s);
    for(int i=0;i<=idx;i++)dp[i][0]=sq[i];
    for(int j=1;j<=20;j++)
    {
        for(int i=0;i+(1<<j)<=idx+1;i++)
        {//这里一定要考虑区间的边界，要满足超出区间右侧一位
            if(d[dp[i][j-1]]<d[dp[i+(1<<j-1)][j-1]])
                dp[i][j]=dp[i][j-1];

            else
                dp[i][j]=dp[i+(1<<j-1)][j-1];
        }
    }
    has[s]=idx;
    for(int i=0;i<m;i++)
    {
        cin >> u >> v;
        cout << lca(has[u],has[v]) << endl;
    }
    return 0;
}
```

- tarjan 算法

```cpp
//tarjan离线算法实现LCA
const int N=5000005;
int arr[N];
int find(int x)
{
    return x==arr[x]?x:arr[x]=find(arr[x]);
}
//这里的merge不能用秩平衡定理
void merge(int x,int y){
    x = find(x);
    y = find(y);
    arr[x] = y;
    return;
}
//兄弟链表法存储所有边
int e[2*N],ne[2*N],h[N],idx;
inline void add(int a,int b)
{
    e[idx] = b,ne[idx] = h[a],h[a] = idx++;
}
//用来存储所有的查询以及查询的序号
vector<int> q[N];
vector<int> qi[N];
void add_query(int u,int v,int id)
{
    q[u].push_back(v);
    q[v].push_back(u);
    qi[u].push_back(id);
    qi[v].push_back(id);
    return;
}
int st[N];
int ans[N];
void DFS(int k)
{
    //初次访问
    st[k] = 1;
    for(int i=h[k];i;i=ne[i])
    {
        if(st[e[i]]) continue;
        DFS(e[i]);
        //注意这里只能单向合并，可以直接设置父节点
        merge(e[i],k);
    }
    //这里一定要注意遍历顺序是后根序。
    //查看是否已经有可以求解的
    for(int i=0;i<q[k].size();i++)
    {
        if(st[q[k][i]]==2){
            ans[qi[k][i]]=find(q[k][i]);
        }
    }
    //遍历结束
    st[k] = 2;
    return;
}
int main(){
    int n,m,s;
    cin>>n>>m>>s;
    //初始化并查集
    for(int i=1;i<=n;i++)arr[i]=i;
    int u,v;
    for(int i=0;i<n-1;i++)
    {
        cin >> u >> v;
        add(u,v);
        add(v,u);
    }
    for(int i=0;i<m;i++)
    {
        cin >> u >> v;
        add_query(u,v,i);
    }
    DFS(s);
```

```
    for(int i=0;i<m;i++)
    {
        cout << ans[i] << endl;
    }
}

```

树的重心

- 对于树上的每一个点,计算所有子树中的最大的子树结点数,这个值最小的点就是树的重心.(这里所有的子树都是指无根树的子树,即包括向上的那棵子树,并且不包括整棵树自身).

> 性质 **1** .树的重心如果不唯一,则至多有两个,且这两个重心相邻. **2** .以树的重心为根时,所有子树的大小都不超过整棵树大小的一半. **3.** 树中的所有点到某个点的距离中,到重心的距离和是最小的;如果有两个重心,那么到他们的距离和一样. **4.** 把两棵树通过一条边相连得到的一棵新的树,那么新的树的重心在连接原来两棵树的重心的路径上. **5** .在一棵树上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.

```
// 这份代码默认节点编号从 1 开始，即 i ∈ [1,n]
int size[MAXN],  // 这个节点的「大小」（所有子树上节点数 + 该节点）
    weight[MAXN],  // 这个节点的「重量」
    centroid[2];   // 用于记录树的重心（存的是节点编号）
void dfs(int cur, int fa)
{  // cur 表示当前节点 (current)
  size[cur] = 1;
  weight[cur] = 0;
  for (int i = head[cur]; i != -1; i = e[i].nxt)
  {
        if (e[i].to != fa)
        {  // e[i].to 表示这条有向边所通向的节点。防止访问到父节点,成环死循环
          dfs(e[i].to, cur);
          size[cur] += size[e[i].to];
          weight[cur] = max(weight[cur], size[e[i].to]);
        }
  }
  weight[cur] = max(weight[cur], n - size[cur]);
  if (weight[cur] <= n / 2) // 依照树的重心的定义统计,最多有两种这种点
    centroid[centroid[0] != 0] = cur;//防止有两个重心
}
```

拓扑排序

> - 主要统计入度和出度的关系,**有向无环图一定能拓扑排序**

```
bool topsort()//d 表示入度,q数组存的是拓扑序
{
    int hh = 0,tt = -1;
    for(int i=1;i<=n;i++)
    {
        if(!d[i])
            q[++tt] = i;
    }//先将入度为0的点入队
    while(hh <= tt)
    {
        int t = q[hh++];
        for(int i=h[t];i!=-1;i=ne[i])
        {
            int j = e[i];
            d[j]--;//不断更新入度出度和队列中的点
            if(d[j] == 0) q[++tt] = j;
        }
    }
    return tt == n-1;//判断是否全都入队了，由此判断是否存在拓扑结构
}   //tt从0开始所以一共n-1     有向无环图
```

最短路

单源正权最短路 --- Dijkstra

> - 堆优化的 Dijkstra 算法,-mlogn,找到目前权值最小的点,用这个点更新其他点到根节点的权值,直至堆为空.

```
int dijkstra()
{
    memset(dist,0x3f,sizeof dist);
```

```
    dist[1]=0;
    priority_queue<PII,vector<PII>,greater<PII>> heap;//小根堆
    heap.push({0,1});//将起点放进去
    while(heap.size())//堆是空的
    {
        auto t = heap.top();//小根堆，堆顶就是最小值
        heap.pop();
        int ver = t.second,distance = t.first;
        if(st[ver]) continue;//这个点是冗余备份，即这个点已经处理过了
        st[ver] = 1;
        for(int i=h[ver];i!=-1;i=ne[i])
        {
            int j = e[i];
            if(dist[j]>distance+w[i])
            {
                dist[j] = distance + w[i];
                heap.push({dist[j],j});
            }
        }
    }
    if(dist[n] == 0x3f3f3f3f) return -1;//图是不连通的
    return dist[n];
}
```

单源负权最短路 ---- Spfa

```
int spfa()//st 数组和Dijk 的区别是Dijk只遍历一遍,Spfa是判断当前点是否在队列里,需要时时更新
{
    memset(dist,0x3f,sizeof dist);//判断负环时不必将距离初始化为0x3f3f3f3f;判断最短路需要初始化,并且不需要cnt数组
    queue<int> q;
    for(int i=1;i<=n;i++)//最短路不需要,只需要找起点入队,更新st[起点]即可
    {//判断存在负环，但不一定是1经过负环，任意一个起点都有可能
        st[i] = 1;
        q.push(i);
    }
    while(q.size())
    {
        int t = q.front();//代表从t到j点有边
        q.pop();
        st[t] = 0;
        for(int i=h[t];i!=-1;i=ne[i])
        {
            int j = e[i];
            if(dist[j]>dist[t]+w[i])
            {
                dist[j] = dist[t]+w[i];
                cnt[j] = cnt[t] + 1;
                if(cnt[j]>=n) return 1;//存在负环,判断最短路的时候不能加
                if(!st[j])
                {
                    q.push(j);
                    st[j] = 1;
                }
            }
        }
    }
    return 0;//表示不存在负环
}
```

多源最短路 -- floyd

```
void floyd()
{
    for(int k=1;k<=n;k++)
        for(int i=1;i<=n;i++)
            for(int j=1;j<=n;j++)//这样的循环顺序可以保证后面的数据更新是
                d[i][j]=min(d[i][j],d[i][k]+d[k][j]);//用到的前面的数据是更新过的
}//读入的时候需要去重边,保存较小的边
if(d[a][b]>INF/2) puts("impossible");//图不连通,判断的时候不能直接用INF,可能有负权边
```

最小生成树

- 最小生成树的唯一性：如果一条边 **不在最小生成树的边集中**，并且可以替换与其**权值相同，并且在最小生成树边集**的另一条边，那么这个最小生成树就是不唯一的.先统计当前权值的总个数,然后计算选上的个数,如果未完全选上,则代表枚举到的点之前已经有联通块,加上此权值的边会成环,不必全选但是路径不唯一,即是最小生成树不唯一

```
const int N = 200010, mod = 1e9+7;
int T, n, m;
struct node{
    int x, y, w;
}a[N];
int ans, pre[N];
int sum;
bool cmp(node a, node b){
    return a.w < b.w;
}
int find(int x){
    if(pre[x] != x) pre[x] = find(pre[x]);
    return pre[x];
}
int kruskal()
{
    sort(a+1, a+m+1, cmp);
    for(int i=1;i<=m;i++)
    {
        int r = i;
        while(r <= m && a[r].w == a[i].w) r++;//记录当前枚举到的权值
        r --;
        int cnt = 0; //记录可拿边的个数
        for(int j=i;j<=r;j++)
        {
            int x = a[j].x, y = a[j].y, w = a[j].w;
            if(find(x) != find(y)) cnt++; //可拿边个数++ 此时不合并
        }
        for(int j=i;j<=r;j++)
        {
            int x = a[j].x, y = a[j].y, w = a[j].w;
            if(find(x) != find(y)) pre[find(x)] = find(y), sum += w, cnt --; //用掉了，cnt--
        }
        i = r;
        if(cnt) return 0; //最后还剩余可拿边，最小生成树不唯一
    }
    return 1;
}
signed main(){
    Ios;
    cin >> T;
    while(T --)
    {
        cin >> n >> m;
        ans = 0, sum = 0;
        for(int i=1;i<=n;i++) pre[i] = i;
        for(int i=1;i<=m;i++)
        {
            int x, y, w;cin >> x >> y >> w;
            a[i] = {x, y, w};
        }
        if(!kruskal()) cout << "Not Unique!\n" ;
        else cout << sum << endl;
    }
    return 0;
}
```

Prim

> - o(n^2)-稠密图-一般是无向图,边多点少,找到集合外距离最近的点用t更新其他点到集合的距离

```
int prim()//g数组是邻接矩阵,存的是两点之间的最小边权
{
    memset(dist,0x3f,sizeof dist);
    int res = 0;//最小生成树里面所有边的长度之和
    for(int i=0;i<n;i++)
    {
        int t = -1;
        for(int j=1;j<=n;j++)
            if(!st[j] && (t==-1||dist[t]>dist[j]))
                t = j;
```

```
        if(i && dist[t] == INF) return INF;//不是第一个点，并且距离集合最近的点是正无穷，代表不连通
        if(i) res += dist[t];//只要不是第一个点加上距离    dist表示这个点到集合的问题
        for(int j=1;j<=n;j++)//先加和在更新，防止自环
            dist[j] = min(dist[j],g[t][j]);//到集合的距离
        st[t] = true;
    }
    return res;//t == INF 不连通的图
}
```

Kruskal

> - 稀疏图-先对边递增排序-o（mlongn),点多边少

```
struct Edge//结构体存储所有边
{
    int a,b,w;
    bool operator< (const Edge &W) const
    {
        return w < W.w;//重载小于号，方便排序
    }
}edges[N];
int find(int x)
{
    if(p[x] != x) p[x] = find(p[x]);
    return p[x];
}
//核心代码
sort(edges,edges+m);
for(int i=1;i<=n;i++) p[i] = i;
int res = 0,cnt=0;
for(int i=0;i<m;i++)
{
    int a = edges[i].a,b = edges[i].b,w = edges[i].w;
    a = find(a),b = find(b);//并查集找根节点
    if(a!=b) //a b 未连通
    {
        p[a] = b;//合并
        res += w;
        cnt ++;//存了多少条边
    }
}//cnt < n - 1,代表的是图不连通,并查集里面的点小于n个,根节点未计入
```

次小生成树

- **非严格次小生成树** :在无向图中,边权最小的满足边权和 **大于等于** 最小生成树边权的生成树,依次枚举未被选上的边(u->v),找到这条边为端点的最小生成树路径内的边权最大的边,用为选中的边代替权值最大的边,
  $M` = M + w - w`$. 对所有替换到的答案 $M`$ 取最小值即可,
- 严格次小生成树 : 边权最小的满足边权和 **严格大于** 最小生成树边权和的生成树.替换的边要严格大于路径中最大边权的权值.
-
```
const int N = 510,M = 10010;
int n,m;
struct Edge
{
    int a,b,w;
    bool f;//表示是不是非树边
    bool operator<(const Edge &t)const
    {
        return w < t.w;
    }
}edge[M];
int p[N],dist[N][N];
int h[N],e[N*2],w[N*2],ne[N*2],idx;//用来建树
void add(int a,itn b,int w)
{
    e[dix] = b,w[idx] = c,ne[idx] = h[a],h[a] = idx ++;
}
int find(int x)
{
    if(x != p[x])
        p[x] = find(p[x]);
    return p[x];
}
void dfs(int u,int fa,int maxd,int d[])//maxd存的是从根节点到当前点的边权最大值
{
```

```
    d[u] = maxd;
    for(int i=h[u];~i;i=ne[i])
    {
        int j = e[i];
        if(j != fa)
        {
            dfs(j,u,max(maxd,w[i]),d);//维护的时候更新一下最大值即可
        }
    }
}
int main()
{
    _;
    cin >> n >> m;
    memset(h,-1,sizeof h);
    for(int i=0;i<m;i++)
    {
        int a,b,w;
        cin >> a >> b >> w;
        edge[i] = {a,b,w};
    }
    sort(edge,edge + m);
    for(int i=1;i<=n;i++) p[i] = i;
    ll sum = 0;
    for(int i=0;i<m;i++)
    {
        int a = edge[i].a,b = edge[i].b,w = edge[i].w;
        int pa = find(a),pb = find(b);
        if(pa != pb)
        {
            p[pa] = pb;
            sum += w;
            add(a,b,w),add(b,a,w);//建树，无向边
            edge[i].f = 1;//标记是树边
        }
    }
    for(int i=1;i<=n;i++)
        dfs(i,-1,0,dist[i]);//传入父节点是因为是双向边防止死循环
    for(int i=0;i<m;i++)
    {
        if(!edge[i].f)//是非树边
        {
            int a = edge[i].a,b = edge[i].b,w = edge[i].w;
            if(w > dist[a][b])
            {
                res = min(res,sum + w - dist[a][b]);
            }
        }
    }
    cout << res << endl;
    return 0;
}//4 4   1 2 100   2 4 200   2 3 250   3 4 100  ----> 450
```

> 优化 : 求解方法基本一致 : 使用倍增的思路维护,差别在于如果找到的最大边权和当前枚举的边权一致,使用严格次大值来替换枚举到的权值.

```
const int INF = 0x3fffffff;
const long long INF64 = 0x3fffffffffffffffLL;
struct Edge {
  int u, v, val;
  bool operator<(const Edge &other) const { return val < other.val; }
};
Edge e[300010];
bool used[300010];
int n, m;
long long sum;
class Tr {
 private:
  struct Edge {
    int to, nxt, val;
  } e[600010];
  int cnt, head[100010];
  int pnt[100010][22];
  int dpth[100010];
  // 到祖先的路径上边权最大的边
```

```cpp
  int maxx[100010][22];
  // 到祖先的路径上边权次大的边，若不存在则为 -INF
  int minn[100010][22];
 public:
  void addedge(int u, int v, int val) {
    e[++cnt] = (Edge){v, head[u], val};
    head[u] = cnt;
  }
  void insedge(int u, int v, int val) {//无向图
    addedge(u, v, val);
    addedge(v, u, val);
  }
  void dfs(int now, int fa) {
    dpth[now] = dpth[fa] + 1;
    pnt[now][0] = fa;
    minn[now][0] = -INF;
    for (int i = 1; (1 << i) <= dpth[now]; i++) {
      pnt[now][i] = pnt[pnt[now][i - 1]][i - 1];
      int kk[4] = {maxx[now][i - 1], maxx[pnt[now][i - 1]][i - 1],
                   minn[now][i - 1], minn[pnt[now][i - 1]][i - 1]};
      // 从四个值中取得最大值
      std::sort(kk, kk + 4);
      maxx[now][i] = kk[3];
      // 取得严格次大值
      int ptr = 2;
      while (ptr >= 0 && kk[ptr] == kk[3]) ptr--;
      minn[now][i] = (ptr == -1 ? -INF : kk[ptr]);
    }
    for (int i = head[now]; i; i = e[i].nxt) {
      if (e[i].to != fa) {
        maxx[e[i].to][0] = e[i].val;
        dfs(e[i].to, now);
      }
    }
  }
  int lca(int a, int b) {
    if (dpth[a] < dpth[b]) std::swap(a, b);
    for (int i = 21; i >= 0; i--)
      if (dpth[pnt[a][i]] >= dpth[b]) a = pnt[a][i];
    if (a == b) return a;
    for (int i = 21; i >= 0; i--) {
      if (pnt[a][i] != pnt[b][i]) {
        a = pnt[a][i];
        b = pnt[b][i];
      }
    }
    return pnt[a][0];
  }
  int query(int a, int b, int val) {
    int res = -INF;
    for (int i = 21; i >= 0; i--) {
      if (dpth[pnt[a][i]] >= dpth[b]) {
        if (val != maxx[a][i])
          res = std::max(res, maxx[a][i]);
        else
          res = std::max(res, minn[a][i]);
        a = pnt[a][i];
      }
    }
    return res;
  }
} tr;
int fa[100010];
int find(int x) { return fa[x] == x ? x : fa[x] = find(fa[x]); }
void Kruskal() {
  int tot = 0;
  std::sort(e + 1, e + m + 1);
  for (int i = 1; i <= n; i++) fa[i] = i;

  for (int i = 1; i <= m; i++) {
    int a = find(e[i].u);
    int b = find(e[i].v);
    if (a != b) {
      fa[a] = b;
      tot++;
      tr.insedge(e[i].u, e[i].v, e[i].val);
```

```
      sum += e[i].val;
      used[i] = 1;
    }
    if (tot == n - 1) break;
  }
}
int main() {
    _;
  std::cin >> n >> m;
  for (int i = 1; i <= m; i++) {
    int u, v, val;
    std::cin >> u >> v >> val;
    e[i] = (Edge){u, v, val};
  }
  Kruskal();
  long long ans = INF64;
  tr.dfs(1, 0);
  for (int i = 1; i <= m; i++) {
    if (!used[i]) {
      int _lca = tr.lca(e[i].u, e[i].v);
      // 找到路径上不等于 e[i].val 的最大边权
      long long tmpa = tr.query(e[i].u, _lca, e[i].val);
      long long tmpb = tr.query(e[i].v, _lca, e[i].val);
      // 这样的边可能不存在，只在这样的边存在时更新答案
      if (std::max(tmpa, tmpb) > -INF)
        ans = std::min(ans, sum - std::max(tmpa, tmpb) + e[i].val);
    }
  }
  std::cout << (ans == INF64 ? -1 : ans) << '\n';  // 次小生成树不存在时输出 -1
  return 0;
}
```

斯坦纳树

- 与最小生成树的区别在于可以外加点使得所有点连通的距离和最小
- **定理：**对于 a,b,c 三个点加入额外点 P 使 a+b+c 最小,(a,b,c分别为PA,PB,PC), 若三角形ABC内角均小于 120°, 那么 P 就是使边$PA.PB,PC$两两之间成 120° 的点的角都是 120° 的点, 如果三角形ABC的一个角大于或等于 120° 那么 P 点与这个点重合.

> 问题推广：数学上表述为：给定n个点 $A_1, A_2, \ldots A_n$, 试求连接此 n 个点, 总长最短的直线连接系统, 并且任意两点都可以由系统中的直线段组成的折线连接起来. 此新问题称为 *斯坦纳树问题* . 在给定 n 个点的情形, 最多将有 $n-2$ 个复接点(斯坦纳点). 过每一个斯坦纳点, 至多有三条边通过. 若为三条边, 则他们两两相交成120°角, 若为两条边, 则此斯坦纳点必为某一一给定的点, 且此两条边交成的角比大于或等于120°

差分约束

- 对于最短路问题,队列的作用是将遍历的点存起来,栈也可以实现这个功能,队列是先进先出,栈是先进后出,对于存在一个负环的问题,由于栈的先进后出,会使此后遍历到的点尽可能是环内的点,减少不必要遍历,同时**当点入队的次数超过 (2 * 点数) 次时,一般认为图中含有环**(一般是不断超时加的优化),某个点入队次数超过 n 次,也被认为含环,最短路找负环,最长路找正环.
-
```
//最短路问题更新
if(dist[j] > dist[t] + w[i]) // j = e[i]
{
    dist[j] = dist[t] + w[i];
    cnt[j] = cnt[t] + 1;
    if(cnt[j] >= n)
        return true;
    if(!st[j])
    {
        q[tt ++] = j;
        if(tt == N) tt = 0;
        st[j] = true;
    }
}
// 最长路更新
if(dist[j] < dist[t] + wf[t] - mid * wt[i]) // j = e[i]
{//最长路：边更新变大，与最短路相反
    dist[j] = dist[t] + wf[t] - mid * wt[i];
    cnt[j] = cnt[t] + 1;
    if(cnt[j] >= n) return true;
    if(!st[j])
    {
        q[tt ++] = j;
        if(tt == N) tt = 0;
        st[j] = true;
    }
```

```
}
//最短路和最长路的区别主要在于更新的时候大小于号的区别
```

- 查分约束 : 用来求不等式的可行解,形如 : $X_i \leq X_j + C_k$的方程组的一组可行解, **源点** 需要满足的条件 : 从源点出发,一定可以走到所有的边. 如果无解一定是走到了 $X_i < X_i$ 的情况 $\Longrightarrow$ 存在负环(
  1.先将每个不等式 $X_i \leq X_j + C_K$ 转换为一条从 X_i走到 X_j 长度为 C_K 的一条边 2.找到一个超级源点, 使得该源点一定可以遍历到所有的边 3.从源点求一遍单源最短路, ①如果存在负环, 则原不等式无解, ②如果没有负环, 则$dist$ 就是原不等式的一个可行解)
- 求最值(指的是每一个变量的最值) **1. 求最小值**,则对应求**最长路** **2.求最大值**,则对应求**最短路**. (主要转换为 $X_i \leq c$ 其中 c 是一个常数, 这类的不等式.)
- 方法 : 建立一个超级源点 0 ,然后建立 $0 \rightarrow i$, 长度是 c 的边即可. 以求X_i的*最大值*为例 : 求所有从X_i出发, 构成的不等式链$X_i \leq X_j + C_1 \leq X_k + C_2 + C_1 \leq \cdots \leq C_1 + C_2 + \ldots$ 所计算出的上界, 最终X_i的最大值等于所有上界的**最小值**,每一条链都转换为一条最长路,求最长路的最小值.结果便是X_i的**最大值**
-

```
int q[N],cnt[N];//cnt用来求正环,q是队列
bool st[N];//是否在队列里
void add(int a,int b,int c)
{
    e[idx] = b,w[idx] = c,ne[idx] = h[a],h[a] = idx++;
}
bool spfa()
{
    int hh = 0,tt = 1;
    memset(dist,-0x3f,sizeof dist);//最长路，需要初始化为负无穷
    dist[0] = 0;//超级源点
    q[0] = 0;
    st[0] = true;
    while(hh != tt)//虽然栈存求负环spfa会更快，但是在求一般的问题时，栈会十分么慢，最长路这种无环的时候更新从后往前会造成许多浪费，本题队列超时，需改为栈尝试
    {
        //int t = q[hh ++];
        int t = q[-- tt];
        //if(hh == N) hh = 0;
        st[t] = false;
        for(int i=h[t];~i;i=ne[i])
        {
            int j = e[i];
            if(dist[j] < dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                cnt[j] =  cnt[t] + 1;
                if(cnt[j] >= n + 1)//包括 0 号点，一共 n + 1 个点
                    return false;
                if(!st[j])
                {
                    q[tt ++] = j;
                    //if(tt == N) tt = 0;
                    st[j] = true;
                }
            }
        }
    }
    return true;
}
int main()
{
    _;
    cin >> n >> m;
    memset(h,-1,sizeof h);
    while(m --)
    {
        int x,a,b;
        cin >> x >> a >> b;
        if(x == 1)
            add(b,a,0),add(a,b,0);
        else if(x == 2)//从 a 向 b 连一条长度为 1 的边
            add(a,b,1);
        else if(x == 3)
            add(b,a,0);//从 b 向 a 连一条长度为 0 的边
        else if(x == 4)
            add(b,a,1);
        else add(a,b,0);
    }
```

```
    for(int i=1;i<=n;i++) add(0,i,1);//源点到每个点的距离为 1 ，源点的值为0，每个人大于1
    if(!spfa())
        cout << -1 << endl;//有负环无解
    else
    {
        ll res = 0;
        for(int i=1;i<=n;i++) res += dist[i];
        cout << res << endl;
    }
}//5 7  1 1 2  2 3 2  4 4 1  3 4 5  5 4 5  2 3 5  4 5 1  ---> 11
```

-

二分图

染色法

> - 染色法判断是不是二分图,二分图一定不含奇数环,不含奇数环的是二分图

```
//链式前向星建边,略,无向图,边是点的二倍
bool dfs(int u,int c)
{
    color[u] = c;//记录当前点的颜色是c
    for(int i = h[u];i!=-1;i=ne[i])//遍历点的领点
    {
        int j = e[i];
        if(!color[j])//没有染色
        {
            if(!dfs(j,3-c)) return false;
        }//没有染色，染成另外一种颜色1或2
        else if(color[j]==c)
            return false;//边的端点颜色一样
    }
    return true;
}
bool flag = true;//表示染色是不是有矛盾
for(int i=1;i<=n;i++)//染色
    if(!color[i])
    {
        if(!dfs(i,1))
        {
            flag = false;
            break;
        }
    }
//flag = 1,表示是二分图,否则不是二分图
```

匈牙利算法

> - 求二分图最大匹配数,如果现在匹配的两个点都是单独的,就匹配在一起,否则就遍历其中一个点的其他匹配可否更换,,如果可以更换就更换,否则匹配失败

```
int match[N];//右边的相对应的点//match[j]=a,表示女孩j的现有配对男友是a
bool st[N];//st[]数组我称为临时预定数组，st[j]=a表示一轮模拟匹配中，女孩j被男孩a预定了。
bool find(int x)//这个函数的作用是用来判断,如果加入x来参与模拟配对,会不会使匹配数增多
{    //遍历自己喜欢的女孩
    for(int i=h[x];i!=-1;i=ne[i])//左半部分
    {
        int j = e[i];
        if(!st[j])//如果在这一轮模拟匹配中,这个女孩尚未被遍历
        {
            st[j] = true;//更新状态
            if(match[j]==0||find(match[j]))//右半部分对应的点未匹配左半部分或者右半部分匹配的左边有其他选择，
            { //如果女孩j没有男朋友，或者她原来的男朋友能够预定其它喜欢的女孩。配对成功,更新match
                match[j] = x;
                return true;
            }
        }
    }//自己中意的全部都被预定了。配对失败。
    return false;
}
for(int i=1;i<=n1;i++)
{//因为每次模拟匹配的预定情况都是不一样的所以每轮模拟都要初始化
    memset(st,false,sizeof st);//将右半部分清空，保证只匹配一次,macth不能清楚,所以不会影响匹配结果 n
```

```
    if(find(i)) res++;
}
```

数学知识

素数筛法

线性筛

```
void get_primes(int n)
{
    for(int i=1;i<=n;i++)
    {
        if(!st[i])
        {
            primes[cnt++] = i;
            for(int j=i+i;j<=n;j+=i) st[j] = true;
        }
    }
}
```

欧拉筛

```
void get_primes(int n)
{
    for(int i=2;i<=n;i++)
    {
        if(!st[i]) primes[cnt++]=i;
        for(int j=0;primes[j]<=n/i;j++)
        {
            st[primes[j]*i] = true;//把每个合数用最小质因子筛掉即可
            if(i%primes[j]==0) break;//primes[j]一定是i的最小质因子
        }
    }
}
```

约数

约数个数

- 对于一个大于1的正整数n可以分解质因数$\mathbf{n}=\prod_{i=1}^{k}\mathbf{p}_{i}^{\alpha_i}=\mathbf{p}_{1}^{a_1}*\mathbf{p}_{2}^{a_2}\cdots\mathbf{p}_{k}^{a_k}$,则约数的个数为$\mathrm{f(n)}=\prod_{i=1}^{t}(a_i+1)=(a_1+1)(a_2+1)\cdots($
- 其中$\alpha_1,\alpha_2\cdots$是$P_1,P_2\cdots P_k$的指数

```
unordered_map<int,int> primes;//哈希表,存素数,将大值映射为小值
while(n--)
{
    int x;
    cin >> x;
    for(int i=2;i<=x/i;i++)//试除法判断这个数的质因子有什么,存入primes里面
        while(x % i ==  0)
        {
            x /= i;
            primes[i] ++;//存质因子个数
        }
    if(x > 1) primes[x] ++;//这个约数较大超过了sqrt(x)
}
ll res = 1;
for(auto prime : primes) res = res * (prime.second + 1) % mod;//计算约数个数
```

约数求和

- 对于一个大于1的正整数n可以分解质因数$\mathbf{n}=\prod_{i=1}^{k}\mathbf{p}_{i}^{\alpha_i}=\mathbf{p}_{1}^{a_1}*\mathbf{p}_{2}^{a_2}\cdots\mathbf{p}_{k}^{a_k}$,则余数之和$sum$等于
- $sum=((P_1^0+P_1^1+P_1^2\cdots+P_1^{a_1})*(P_2^0+P_2^1+P_2^2\cdots+P_2^{a_2})*\cdots*(P_k^0+P_k^1+P_k^2\cdots+P_k^{a_k}))$

```
unordered_map<int,int> primes;//哈希表
while(n--)
{
    int x;
    cin >> x;
```

```
    for(int i=2;i<=x/i;i++)
        while(x%i==0)
        {
            x /= i;
            primes[i]++;//存质因子个数
        }
    if(x > 1) primes[x]++;//这个约数较大超过了sqrt(x)
}
ll res = 1;
for(auto prime : primes)
{
    int p = prime.first,a = prime.second;
    ll t = 1;
    while(a--)
        t = (t * p + 1) % mod;//求得是对于每一个质因子的部分和
    res = res * t % mod;//结果是所有部分和 t 的乘积.
}
```

欧拉函数

- 定义 : 对于一个正整数 n,n 的欧拉函数 $\Phi(n)$,表示小于等于 n 与 n 互质的正整数的个数.
- 性质 1 : 如果 n 是质数,那么 $\Phi(n) = n - 1$,因为只有 n 本身与它不互质.
- 性质 2 : 如果 p 和 q 是质数,那么 $\Phi(p * q) = \Phi(p) * \Phi(q) = (p - 1) * (q - 1)$.
- 性质 3 : 如果 p 是质数,那么$\Phi(\mathbf{p^k}) = \mathbf{p^k - p^{k-1}}(p, 2p, 3p \cdots p^{k-1}p$一共$p^{k-1}$个能够整除$p^k$的,所以答案为$p^k - p^{k-1})$
- 性质 4 :对于任意正整数$n, \Phi(\mathbf{n}) = \mathbf{n}(\mathbf{1} - \frac{1}{\mathbf{p_1}})(\mathbf{1} - \frac{1}{\mathbf{p_2}})(\mathbf{1} - \frac{1}{\mathbf{p_3}}) \cdot \cdot \cdot \cdot \cdot (\mathbf{1} - \frac{1}{\mathbf{p_n}})$
- 其余性质 : 若\ a 为质数,b mod a = 0,则
  $\Phi(a * b) = \Phi(b) * a$, 若 a 和 b 互质, $\Phi(a * b) = \Phi(a) * \Phi(b)$, 若 n 为一个正整数, 那么$\Sigma_{\mathrm{d|n}} = \mathbf{n}$, 其中 $d \mid n$ 表示 d 整除 n

```
for(int i=2;i<=n;i++)
{//prime[0]表示素数个数
    if(!vis[i])//没有被访问，也就是没有被筛掉,说明是素数
    {
        vis[i] = 1;
        prime[++ prime[0]]=i;
        phi[i]=i-1;
    }
    for(int j=1;j<=prime[0]&&i*prime[j]<=n;j++)
    {
        vis[i * prime[j]]=true;
        if(i % prime[j] == 0)//a % b == 0，那么phi[a * b] = b * phi[a]
        {
            phi[i * prime[j]] = phi[i] * prime[j];
            break;
        }
        else phi[i * prime[j]] = phi[i] * (prime[j] - 1);//两者互素,素数的欧拉函数是 n-1
    }
}
```

快速幂

```
int qmi(int a,int k,int p)
{
    int res=1;
    while(k)
    {
        if(k&1) res = (ll)res * a % p;// k & 1 -> 将k换为二进制且求最后一位0或1
        k >>= 1;//右移运算符，二进制删除最后一位
        a = (ll)a * a % p;//实现平方操作
    }
    return res;
}
//如果数据范围比较大,可能会爆longlong,此时需要优化或者使用__int128
ll qmul(ll a,ll k,ll b) //或者使用 __int128
{
    ll res = 0;
    while(k)
    {
        if(k & 1) res = (res + a) % b;
        a = (a + a) % b;
        k >>= 1;
    }
    return res ;
```

```
}
ll qmi(ll a,ll k,ll b)//这题 L 的范围太大，可能会爆 ll ,len(L) = 9,如果数十分大，可能爆 18 ll
{
    ll res = 1;
    while(k)
    {
        if(k & 1)
            res = qmul(res,a,b);
        a = qmul(a,a,b);
        k >>= 1;
    }
    return res;
}
```

逆元

> - 逆元就是在 mod 的意义下,不能直接除以一个数,而是要乘以它的逆元,比如
> $a * b \equiv 1(mod\ p)$, 那么a, b互为模 n 意义下的逆元, 例如要计算 x/a 就可以改成 $x\ *\ b\ \%\ p$

DP

简介

基础类DP

记忆化搜索

背包DP

区间DP

DAG上的DP

树形DP

基础

- 没有上司的舞会 一个人去他的上司不能去,求最大开心指数

```
void dfs(int u)
{
    dp[u][1] = happy[u];
    for(int i=h[u];~i;i=ne[i])
    {
        int j = e[i];
        dfs(j);
        dp[u][1] += dp[j][0];
        dp[u][0] += max(dp[j][1],dp[j][0]);

    }
}//注意建边的顺序,add模板的建边是后传入是先传入的儿子
cout << max(dp[root][1],dp[root][0]) << endl;//root是入度为 0 的点
```

- HDU 2196 Computer POI FAR-FarmCraft

树上背包

- 树上背包就是背包问题和树形DP的结合,例如 洛谷CTSC1997选课 ,有 n 门课程,ai 为对应的学分,每门课有 0 或 1门先修课,有先修课需要先学完先修课,m 门最多的学分 ----> 转换为 01 背包

```
int dp[N][N],w[N],n,m;//f[][][]表示遍历u号点的前i棵树,选了j门课的最大学分,倒序优化一维
vector<int> a[N];
int dfs(int u)
{
    int p = 1;
    dp[u][1] = w[u];
    for(auto L : a[u])//遍历子vector,物品组
```

```
    {
        int si = dfs(L); // 注意下面两重循环的上界和下界
        for(int i=min(p,m+1);i;i--)//体积组倒序
        {// 只考虑已经合并过的子树，以及选的课程数超过 m+1 的状态没有意义
            for(int j=1;j<=si&&i+j<=m+1;j++)//体积组,选了多少门
                dp[u][i + j] = max(dp[u][i + j],dp[u][i] + dp[L][j]);
        }
        p += si;
    }
    return p;
}
dfs(0);
cout << dp[0][m + 1] << endl; //m + 1 是因为加入了 0 号课程,学分是 0
```

- JSOI2018 潜入行动 SDOI2017苹果树 cf Mex Tree

换根DP

- 换根DP是一种用来求树上各点道其他点的距离之和的算法,通常不会指定根节点,并且根节点的变化会对一些值,例如子节点深度和,点权和等产生影响,通常需要两遍DFS,第一次DFS预处理诸如深度,点权和之类的信息,第二次DFS开始进行换根DFS

  > - 用于求出树上所有点为根节点时到其他点的距离之和.
  > - 假设 a为根节点，b为直系子节点，那么对于 b 所在子树对 a 的贡献为 dis[b] + size[b] ,其中 dis[b] 为 **b** 到以 **b** 为根节点的子树中所有点的距离之和。size[b]为以 **b** 为根节点的子树中的点的个数，其实很好理解，就相当于以 b为根节点中的所有路径长度全部 +1，然后就到达了 a 节点。

```
int dis[N];//第一次dfs每个节点到其子节点距离之和
int size[N];//每个节点下子节点个数（包括这个节点本身）
int dp[N];//最终结果
bool vis[N];
vector<int> vec[N];
void dfs(int x)
{
    vis[x] = true;
    int sum = 0;
    for(int i=0;i<vec[x].size();i++)
    {
        int y = vec[x][i];
        if(!vis[y])
        {
            dfs(y);
            sum += size[y];
            dis[x] += dis[y]+size[y];
        }
    }
    size[x] = sum+1;
    return ;
}
```

  > - 换根dp,换根DP的思想就是**把所有与根相连的节点通过一定的操作将其变为根**,依然利用上述节点 a,b,将根节点从 a 换到 b ,dp[b] = dp[a] - size[b] + (n - size[b]),-size[b] 表示从 b 引申出来的 size[b] 条路径长度全部 −1,n−size[b] 表示从 a 引申出来的不包含 b 的其他路径长度全部 +1

```
memset(vis,0,sizeof(vis));//dfs时使用过一次，因此需要清空
dp[a]=dis[a];//此处a指代上面自己找的dfs起点，一般情况下a=1
void Dp(int x)
{
    vis[x] = true;
    for(int i=0;i<vec[x].size();i++)
    {
        int y = vec[x][i];
        if(!vis[y])
        {
            dp[y] = dp[x] - size[y] + n - size[y];
            Dp(y);
        }
    }
}
```

  > - 例题, 给出一个 n 个点的树,请求出一个节点使得以这个点为根节点时,所有节点的深度之和最大. 牛客 : Tree 求深度之和最小

状压DP

数位DP

插头DP

计数DP

动态DP

概率DP

DP优化

其他类DP

计算几何

杂项

字符串

KMP

```
//求next过程,这个板子中将字符串下标转换为 从 1 开始,注意
for(int i=2,j=0;i<=n;i++)
{       //需要特别注意i是从1开始计数的，j是从0开始
    while(j && p[i] != p[j+1]) j = ne[j];//未到起始，且匹配失败，回答前缀的结尾，快速匹配
    if(p[i] == p[j+1]) j++;//匹配成功，前缀结尾终点后移
    ne[i] = j;//更新后缀对应的前缀结尾
}
//KMP匹配过程
for(int i=1,j=0;i<=m;i++)// i 遍历s所有数组
{
    while(j && s[i] != p[j+1]) j = ne[j];// j没有退回起点且匹配失败，  下标从 1 开始，0相当于在空着，起来一个判断作业
    if(s[i] == p[j+1]) j++;
    if(j == n)
    {
        //匹配成功
        cout << i-j << " ";//i从1开始i-n即是开始位置
        j = ne[j];//匹配结束，更新最大后缀对应的前缀结尾，快速匹配前一部分字符
    }
}
//KMP 判断字符串是否是由子字符串重复循环构成
if(len%(len-next[len])==0)//len表示字符串 s 的长度,就是字符串第一位到next[len]位与len-next[len]位到第len位是匹配的
    printf("%d\n",len/(len-next[len]));//判断,len-next[len] 即是重复子字符串的长度
```

字符串哈希 Hash

> - 自动溢出 ull

```
p[0] = 1; // 不要将某个字符串映射为0,所以p[0]初始化为 1,映射的最小值为 1
cin >> n >> m >> str + 1;//n 是字符长度,m 是询问个数,str 是输入字符串,char str[N];
for(int i=1;i<=n;i++)//n 为字符长度
{
    p[i] = p[i-1]*P;// P = 131 或 13331,p,h 数组都是 ull 的数据类型
    h[i] = h[i-1]*P+str[i] - 'A' + 1;//str 是读入的字符串,下标从 1 开始,将字符转换为 1 - 26,减小范围
}
//获取哈希结果
ull get(int l,int r)
{   //公式，相当于将1到l-1这串字符，前移到长度与r对齐，不足补-，这样相减就是l到r的区间哈希
    return h[r] - h[l-1]*p[r-l+1];
}
```

- 二维哈希 : 取两个 base,
- 主要用于统计 b矩阵在 a矩阵中的出现次数

```
const ULL base1 = 13331;
const ULL base2 = 23333;
ULL p1[N],p2[N];
void pre()
{
    p1[0] = p2[0] = 1;//与一维hash一样,不要将字符串映射为 0,p1,p2[0] = 1;
    for(int i=1;i<N;i++)
        p1[i] = p1[i-1] * base1,//记录乘了多少次base
        p2[i] = p2[i-1] * base2;
}
int T,n1,m1,n2,m2;
char a[N][N],b[N][N];
ULL h[N][N];//h[i][j]表示(1,1,i,j)子矩阵的哈希值,类似与前缀和求子矩阵的和
void init()
{//二维哈希的初始化类似与二维前缀和s[i][j] = s[i][j-1] + s[i-1][j] - s[i-1][j-1] + a[i][j],二维哈希乘于base
    for(int i=1;i<=n1;i++)
        for(int j=1;j<=m1;j++)
            h[i][j] = h[i][j-1] * base2 + h[i-1][j] * base1
                - h[i-1][j-1] * base1 * base2 + (a[i][j] - 'A' + 1);
}//减去的值乘两个base是因为加的时候重叠了,两个base都乘了,所以减去的时候要都乘
ULL get_hash(int x,int y,int lx,int ly)
{//计算子矩阵和,联系一维哈希和二维前缀和的计算,
    return  +h[x+lx-1][y+ly-1]//右下角
            -h[x+lx-1][y-1]*p2[ly]
            -h[x-1][y+ly-1]*p1[lx]//p数组存的是计算前缀和时,乘了多少次base
            +h[x-1][y-1]*p1[lx]*p2[ly];//左上角
}
int main()
{
    pre();
    cin >> T;
    while(T--)
    {
        cin >> n1 >> m1;
        for(int i=1;i<=n1;i++) cin >> a[i] + 1;
        cin >> n2 >> m2;
        for(int i=1;i<=n2;i++) cin >> b[i] + 1;
        init();
        ULL t = 0;
        for(int i=1;i<=n2;i++)
            for(int j=1;j<=m2;j++)
                t += (b[i][j]-'A'+1) * p1[n2-i] * p2[m2-j];//注意下标
        int ans=0;
        for(int i=1;i<=n1-n2+1;i++)
            for(int j=1;j<=m1-m2+1;j++)
                if(get_hash(i,j,n2,m2)==t)//i,j表示起点,求的是大小为n,m的子矩阵,判断是否出现过,所以get里面右下角
                    ans++;       //是[x+la-1][y+ly-1],后面的过程类似于二维矩阵求子矩阵和
        cout << ans << endl;
    }
    return 0;

 //如果是一个大的子矩阵求的是若干个小矩阵在这个大矩阵里面有没有出现过,可以   使用二分优化(前提是所有子矩阵的大小一致)
    init();//大矩阵
    for (int i = 1; i <= n - a + 1; i++)
         for (int j = 1; j <= m - b + 1; j++)
              ans[++ct] = get_hash(i, j, a, b);//预处理出来所有的大小为 a * b 的矩阵
     while(q --)
     {
         cin >> str + 1;//处理询问小矩阵
```

```
        for(int i=1;i<=a;i++)
        {
            for(int j=1;j<=b;j++)
            {
                h[i][j] = (ll) h[i][j-1] * base2 + h[i-1][j] * base1
               - h[i-1][j-1] * base1 * base2 + (str[i][j] - 'A' + 1);
            }
        }//binary_search  STL二分,返回值只有0 或 1,不能返回下标,返回类型是bool的
        puts(std::binary_search(ans + 1, ans + ct + 1, h[a][b]) ? "1" : "0");
    }
}
```

- 第二种方法 : 二维哈希先求出每一行的子串,在将行看做点求矩阵哈希

```
for (int i = 1; i <= n; ++i)
    for (int j = 1; j <= m; ++j)
    {
        scanf("%d", &x);
        h[i][j] = ((ull)h[i][j - 1] * 131 + x);
    }
//第一步获得每行子串的哈希表
for (int i = 1; i <= n; ++i)
    for (int j = 1; j <= m; ++j)
        h[i][j] = (h[i][j] + (ull)h[i - 1][j] * 233);
//第二步获得子矩阵的哈希值
```

> - 双哈希,一般不使用自动溢出即是 ull 类型,会被卡,手动取模,双哈希本质上就是两遍一维哈希,不再使用自动溢出

```
const ULL MOD1 = 402653189;
const ULL MOD2 = 805306457;
const ULL BASE1 = 13331;
const ULL BASE2 = 23333;
//备用 模数&base
//const ULL MOD1 = 1610612741;
//const ULL MOD2 = 1222827239;
//const ULL BASE1 = 1331;
//const ULL BASE2 = 131;
ULL p[N][2];
void init()
{
    p[0][0] = p[0][1] = 1;//字符串不映射到 0
    for(int i=1;i<N;i++) p[i][0] = p[i-1][0] * BASE1 % MOD1;
    for(int i=1;i<N;i++) p[i][1] = p[i-1][1] * BASE2 % MOD2;
}//如果字符串长度确定,for 循环不必循环到N,循环到字符串的长度即可
struct myHash
{
    ULL h[N][2];
    void init(char s[],int n = 0)
    {
        n = strlen(s),h[n][0] = h[n][1] = 0;
        for(int i=1;i<=n;i++)//两遍hash
            h[i][0] = (h[i-1][0] * BASE1 % MOD1 + s[i] - 'A' + 1) % MOD1;
        for(int i=1;i<=n;i++)
            h[i][1] = (h[i-1][1] * BASE2 % MOD2 + s[i] - 'A' + 1) % MOD2;
    }
    upr get_hash(int pos,int length)//r=pos,l = l + length +1,
    {//返回哈希的结果,和一维哈希计算结果一样,只是两种哈希模数和基数
        return {
            ((h[pos][0] - h[pos-length][0] * p[length][0] % MOD1) + MOD1) % MOD1,
            ((h[pos][1] - h[pos-length][1] * p[lengt h][1] % MOD2) + MOD2) % MOD2
        };
    }
};
upr make_hash(char s[])
{
    int n = strlen(s);
    upr ans;
    for(int i=n-1;i>=0;i--)
    {
        ans.fi = (ans.fi * BASE1 % MOD1 + s[i] - 'A' + 1) % MOD1;
        ans.se = (ans.se * BASE2 % MOD2 + s[i] - 'A' + 1) % MOD2;
    }
    return ans;
}
```

manacher算法求最大回文串

- 直接暴力复杂度较高,从中间向两边扩展不能确定对称中心是在字符上还是字符中间,所以采用每两个字符加入一个字符例如' #',将所有的字符对称中心转换到符号 '#' 上,结尾 0 和开头使用不同的字符防止匹配上越界

```
string str,s;
int dp[N];
void solve()
{
    cin >> str;
    int n = str.size();
    s += ' ';//防止越界
    for(int i=0;i<n;i++)
    {
        s += '#';//没两个字符中间加上一个辅助字符,强制将对称中心转到字符上,而不是字符中间
        s += str[i];
    }
    s += '#';
    s += '!';//最后再加一个!,防止和开头匹配上导致越界
    int pos = 0,r = 0;//pos是中点,r是回文字符串的最右端点,pos是r对应的中点
    n = s.size();
    for(int i=1;i<n;i++)//pos < i
    {
        if(i < r)//求最右端点,与左端点对称的点和,这段区间已经匹配上了
            dp[i] = min(dp[(pos<<1)-i],r - i);
        else dp[i] = 1;
        while(s[i-dp[i]] == s[i + dp[i]])//因为终点的字符是独特的,所以不用担心越界问题,结尾一定匹配不上
            dp[i] ++;//如果两边的字符能匹配上,dp数组可以匹配的长度加 1
        if(i + dp[i] > r)//匹配的长度大于最大右端点了,更新右端点
            r = i + dp[i],pos = i;//更新右端点和中点
    }
    int ans = 0;
    for(int i=1;i<=n;i++)
        ans = max(ans,dp[i] - 1);//找最长回文
    cout << ans << endl;
}//每个答案减一是因为dp代表的含义是一边但是算上的有#,所以是最大回文,但是结束的时候一定是#,多计算了一个#
signed main()
{
    ios
    int t=1;
    // cin >> t;
    while(t--)
          solve();
    return 0;
}
```