数据结构与算法😍

STL 😵‍💫

详细内容🥰

栈stack 🤩

加入头文件#include< stack > 初始化与vector相同stack< int >a; **常用函数:**

- empty() //判断堆栈是否为空
- pop() //弹出堆栈顶部的元素
- push() //向堆栈顶部添加元素
- size() //返回堆栈中元素的个数
- top() //返回堆栈顶部的元素

队列queue 🤩

加入头文件#include< queue > 初始化与vector相同queue< int >a; **常用函数:**

- back() //返回队列中最后一个元素
- empty() //判断队列是否为空
- front() //返回队列中的第一个元素
- pop() //删除队列的第一个元素
- push() //在队列末尾加入一个元素
- size() //返回队列中元素的个数

向量vector 🤩

- resize(count) //设置vector的大小为count
- resize(count,a) //用a补全vector扩充的大小

iota()函数--->用于填充vector,头文件为#include< numeric >,无返回值

fill()函数--->用于填充vector,头文件为#include< algorithm >,无返回值

> iota(起始位置,结束位置,起始值)

> fill(起始位置,结束位置,填充值)

iota从起始值开始填充，随着长度增长填充值增加"1",填充值不一定是数字，字母的话按照ASCII码增加"1"

fill则不会递增，将填充值填满范围

字符串string🤩

常用函数：

- find()
- to_string() //将基本类型的值转换为字符串
- stoi() //将字符串类型转换为int类型
- substr(pos,size)//从pos开始截取size长度的字符串

数据对pair🤩

pair< 类型1,类型2 >

成员变量是first和second

pair类型定义在#include< utilit >头文件中

字典map🤩

map

map< 类型1,类型2 >

不会存在key相等的情况，那样新的就不会再插入

删除还是使用迭代器删除(迭代器定义 ::iterator iter)，使用key删除有返回值

- clear() //删除所有元素
- find() //查找一个元素,返回迭代器位置
- swap() //交换两个map
- erase() //删除一个元素

字典unordered_map

拥有键值对，用可以来查找value，有find(key)函数

修改value：

```
//直接使用x.value=new_value，在出for循环后值会恢复
for(auto x:unomap)//遍历整个map，输出key及其对应的value值
{
    auto it = umap.find(key) //改
    if(it != umap.end())
        it->second = new_value;
}
```

！！！遍历顺序与输入顺序不一定相同

集合set🤩

加入头文件#include< set > 初始化与vector相同set< int >a; **常用函数:**

- insert() //插入元素
- count() //判断是否存在某元素
- 其余各容器相似

bitset🤩

bitset 中元素只能是1或0,保存在#include< bitset >中

定义: bitset< 序列长度 > 序列名称(初始化元素)，初始化元素若小于长度则右对齐,二维定义：bitset< 行长度 > 序列名称[列长度]

bitset有位运算符(& | ^ ~)

- b.any() b中是否存在置为1的二进制位?

- b.none() b中不存在置为1的二进制位吗?

- b.count() b中置为1的二进制位的个数

- b.size() 访问b中在pos处的二进制位

- b[pos] 访问 b 中在 pos 处的二进制位

- b.test(pos) b中在 pos 处的二进制位是否为 1?
- b.set() 把b中所有二进制位都置为1
- b.set(pos) 把b中在 pos 处的二进制位置为 1
- b.reset() 把 b 中所有二进制位都置为 0
- b.reset(pos)把 b 中在 pos 处的二进制位置为 0
- b.flip() 把 b 中所有二进制位逐位取反
- b.flip(pos) 把 b 中在 pos 处的二进制位取反

双向队列deque🤩

加入头文件#include< deque > 初始化与vector相同deque< int >a; **常用函数:**

- push_back() //在队列的尾部插入元素。
- push_front() //在队列的头部插入元素。
- emplace_back() //与push_back()的作用一样
- emplace_front() //与push_front()的作用一样
- pop_back() //删除队列尾部的元素。
- pop_front() //删除队列头部的元素。
- back() //返回队列尾部元素的引用。
- front() //返回队列头部元素的引用。
- clear() //清空队列中的所有元素。
- empty() //判断队列是否为空。
- size() //返回队列中元素的个数。
- begin() //返回头位置的迭代器
- end() //返回尾+1位置的迭代器
- insert(pos,e) //在指定位置pos插入元素e。 vector也能用

- insert(pos,n,e) //在指定位置pos插入n个元素e
- insert(pos,a,b) //在指定位置pos插入(a,b)区间（左闭右开）的元素
- erase(i) //在指定i位置删除元素
- erase(a,b) //在指定(a,b)区间（左闭右开）删除元素

遍历:

```
deque<int>::iterator it; //迭代器定义
for(it=d.begin();it!=d.end();it++){
    cout<<*it<<" "; //注意*t和for中的!=
}
```

优先级队列priority_queue🤩

在头文件#include< queue >中

- q.top() 访问队首元素
- q.push() 入队
- q.pop() 堆顶（队首）元素出队
- q.size() 队列元素个数
- q.empty() 是否为空

设置优先级

！！！ priority_queue的排列顺序与sort完全相反

> 基本结构：priority_queue< 类型,比较结构cmp > 名
>
> priority_queue< int,vector< int >,greater< int > > 第二个参数是底层存储类型，三个参数中int位置类型要相同；less< int >是最大堆，greater< int >是最小堆

自定义结构体比较

```
//一般写法
struct node {
    int x, y;
    bool operator < (const Point &a) const {//直接传入一个参数，不必要写friend
        return x < a.x;//按x升序排列，x大的在堆顶
```

```
    }
};
```

存储pair类型时默认对first进行降序排列，first相同再对second排列

元组tuple🤩

在头文件#include< tuple >中，可以看作是pair的扩展

> tuple< int,int,string >t--->三元组

赋值：t=make_tuple(a,b,c);

获取元素：int a =get< 0 >(t)--->获得t的第0位

获取元素个数：int count = tuple_size< decltype (t) >::value--->decltype(判断t的类型)，::value指返回一个结果值

解包：tie(a,b,c)=t--->将t中三个元素分别赋值给三个变量

包含在algorithm中的好用函数

- max_element(a.begin(),a.end()) //返回a中最大值的迭代器
- min_element(a.begin(),a.end()) //返回a中最小值的迭代器
- lower_bound(a.begin(),a.end(),x) //返回第一个>=x的值的位置(a有序)，自定义comp为找到false的值
- lower_bound(a.begin(),a.end(),x，less< type >()) //返回第一个>=x的值的位置(a有序)
- lower_bound(a.begin(),a.end(),x，greater< type >()) //返回第一个<=x的值的位置(a有序)
- upper_bound(a.begin(),a.end(),x) //返回第一个>x的值的位置(a有序),其余相同，自定义comp为找到true的值

排序算法😵‍💫

sort排序😋

sort(begin,end)中begin至and为左开右闭，默认升序排列

sort(begin,end,greater< type >())降序排列

自定义排序时返回a>b是降序排列，即返回true值

stable_sort()用法相同，但数据相等时不进行交换，保持稳定性

匿名函数写法：

> [捕捉变量列表] (参数列表) ->返回类型{函数体}

只有一个return时可省略返回类型

> [ ] // 未定义变量.试图在Lambda内使用任何外部变量都是错误的.
>
> [x, &y] // x 按值捕获, y 按引用捕获.
>
> [&] // 用到的任何外部变量都隐式按引用捕获
>
> [=] // 用到的任何外部变量都隐式按值捕获
>
> [&, x] // x显式地按值捕获. 其它变量按引用捕获
>
> [=, &z] // z按引用捕获. 其它变量按值捕获

示例

```
sort(v.begin(), v.end(), [](int a, int b) {  //a和b是列表中元素，可能还是列表
        return a > b;    // 降序排列
    });
```

冒泡排序🤪

时间复杂度O(n^2^)

交换次数等于逆序对数量

```
void BubbleSort(int a[],int left,int right){
    for(int i=left;i<=right;i++){
        for(int j=right-1;i>=i;i--){
            if(a[j]>a[j+1]){
                swap(a[j],a[j+1]);
            }
```

```
            }
        }
    }
```

快速排序🤪

时间复杂度平均O(n*log(n)), 最坏情况O(n^2^)

```
//优化->三数取中法取轴点
int Partition(int a[],int left,int right){
    int i=left;
    int j=right-1;
    int p=a[right];
    while(true){
        while(a[i]<p){
            i++;
        }
        while(a[j]>p && j>left){
            j--;
        }
        if(i>=j){
            break;
        }
        swap(a[i],a[j]);
        i++;
        j++;
    }
    swap(a[i],p);
    return i;
}

void QuickSort(int a[],int left,int right){
    if(left<right){
        int i=Partition(a,left,right);
        QuickSort(a,left,i-1);
        QuickSort(a,i+1,right);
    }
}
```

归并排序🤪

时间复杂度O(n+m)

经典归并排序

```
vector<int> TwoWayMerge(int a[],int lx,int rx,int ly,int ry){
    vector<int>v;
    int i=lx;
    int j=rx;
    while(i<=lx || j<=rx){
        if(j>rx ||(i<=lx && a[i]<=a[j])){
            v.push_back(a[i]);
            i++;
        }
        else{
            v.push_back(a[j]);
            j++;
        }
    }
    return v;
}

//自顶向下
vector<int> MergeSort(int a[],int left,int right){
    vector<int>v;
    if(left<right){
        int m=(left+right)/2;
        MergeSort(a,left,m);
        MergeSort(a,m+1,right);
        v=TwoWayMerge(a,left,m,m+1,right);
    }
    return v;
}

//自下向上
vector<int> MergeSortUp(int a[],int left,int right){
    int len=1;
    int n=right-left+1;
    vector<int>v;
    while(len<n){
        int lx=0;
        int rx;
        int ly;
        int ry;
        while(lx<=right-len){
            int rx=lx+len-1;
            int ly=rx+1;
            int ry=min(ly+len-1,right);
            v=TwoWayMerge(a,lx,rx,ly,ry);  //这一有一点问题，不能直接用v
            lx=ry+1;
        }
    }
    return v;
}
```

感觉快排和归并很像，都像是二分,感觉快排像是自顶至下的归并

链表归并排序

```
link* LinkSort(link* list1,link* list2){
    link* p1=list1;
    link* p2=list2;
    link* pre=NULL;
    while(p2!=NULL){
        while(p1!=NULL && p1->data<p2->data){
            pre=p1;
            p1=p1->next;
        }
        link* temp=p2;
        p2=p2->next;
        temp->next=p1;
        if(pre==NULL){
            list1=temp;
        }
        else{
            pre->next=temp;
        }
        p1=temp;
    }
    return list1;
}
```

求逆序对数量😡

```
int TwoWayInversionCount(int a[],vector<int>v,int l,int m,int r){
    int i=l;
    int j=m+1;
    int count=0;
    while(i<=m || j<=r){
        if(j>r || (i<=m && a[i]<a[j])){
            v.push_back(a[i]);
            i++;
        }
        else{
            v.push_back(a[j]);
            j++;
            count+=(m-i+1);
        }
    }
    return count;
}

int InversionCount(int a[],int l,int r){
    int count=0;
    vector<int>v;
    if(l<r){
        int m=(l+r)/2;
        InversionCount(a,l,m);
        InversionCount(a,m+1,r);
        count+=TwoWayInversionCount(a,v,l,m,r);
    }
```

```
    return count;
}
```

计数排序😜

```
vector<int> CountSort(int a[],int n,int max){
    vector<int>v(0,n);
    int c[max];
    for(int i=0;i<max;i++){
        c[i]=0;
    }
    for(int i=0;i<n;i++){
        c[a[i]]++;
    }
    for(int i=1;i<max;i++){
        c[i]=c[i]+c[i-1];
    }
    for(int i=n-1;i>=0;i--){
        v[c[a[i]]]=a[i];
        c[a[i]]--;
    }
    return v;
}
```

树tree😵‍💫

二叉树🥵

头文件1

```
#include<iostream>
#include<vector>
#include<string>
#include<queue>
using namespace std;
```

构建二叉树结构体1

```
//构建二叉树结构体
typedef struct node{
    int data;
    node*left,*right;
    node(int x):data(x),left(NULL),right(NULL){}  //构造函数
}BT;
```

构建二叉树

```
//构建二叉树
BT* CreatTree(int a){
    node*tree=new node(a);
    /*tree->data=a;
      tree->left=NULL;
      tree->right=NULL;   有构造函数后可省略*/
    return tree;
}
```

前序序列反序列化

```
//前序序列反序列化
int k=-1; //全局变量k
BT* PreDESer(string s){
    k++;
    int n=s.size();
    node*tree=NULL;
    if(k<n){
        int data=int(s[k]); //to_string()使int转string, stoi()是string转int
        if(s[k]!='#'){  //此处#意为空树
            tree=new node(data);
            tree->data=data;
            tree->left=PreDESer(s);
            tree->right=PreDESer(s); //重构左右子树, k决定data
        }
    }
    return tree;
}
```

前中序列反序列化，无"#"😵

```
//前中序列反序列化，无#
BT* PreInDESer(string pre, string inorder,int preIndex,int Instart, int Inend ){
    if(Instart>Inend) return NULL;
    node*tree=new node(int(pre[preIndex]));
    preIndex++;
```

```
    int Index=Instart;
    for(;Index<Inend;Index++){
        if(inorder[Index]==pre[preIndex]) break;
    }
    tree->left=PreInDESer(pre, inorder, preIndex, Instart,  Index-1 );
    tree->right=PreInDESer(pre, inorder, preIndex, Index+1,  Inend );
    return tree;
}
```

删除树

```
//删除树
void remove(BT* tree) {
    if (tree == nullptr) return;
    remove(tree->left);
    remove(tree->right);
    delete tree;
    tree = nullptr;
}
```

前序，中序，后序遍历

```
//前序，中序，后序遍历
void PreOrder(BT*tree){
    if(tree!=NULL){
        cout<<tree->data<<" ";
        PreOrder(tree->left);
        PreOrder(tree->right);
    }
}

void InOrder(BT*tree){
    if(tree!=NULL){
        InOrder(tree->left);
        cout<<tree->data<<" ";
        InOrder(tree->right);
    }
}

void PostOrder(BT*tree){
    if(tree!=NULL){
        PostOrder(tree->left);
        PostOrder(tree->right);
        cout<<tree->data<<" ";
    }
}
```

层序遍历

```
//层序遍历
void Sequence(BT*tree){
    queue<BT*>q;
    q.push(tree);
    while(!q.empty()){
        BT*node_ptr=q.front();
        q.pop();
        if(node_ptr!=NULL){
            cout<<node_ptr->data<<" ";
            q.push(node_ptr->left);
            q.push(node_ptr->right);
        }
    }
}
```

主函数1

```
int main(){
    int a=0;
    BT*tree=CreatTree(a);
    return 0;
}
```

二叉搜索树🥵

头文件2

```
#include<iostream>
#include<vector>
#include<string>
using namespace std;
```

构建二叉树结构体2

```
//构建二叉树结构体
typedef struct node{
    int data;
    node*left,*right;
    node(int x):data(x),left(NULL),right(NULL){}  //构造函数
}BT;
```

查找

```
//查找
BT*Search(BT*bst ,int key){
    if(bst==NULL || bst->data==key) return bst;
    else if(bst->data < key) return Search(bst->left,key);
    else return Search(bst->right,key);
}
```

插入1

```
//插入
BT* Insert(BT* bst, int key){
    if(bst==NULL) bst=new node(key);
    else if(key<bst->data) bst->left=Insert(bst->left,key);
    else bst->right=Insert(bst->right,key);
    return bst;
}
```

删除1 😰

```
//删除
BT* Removal(BT*bst, int key){
    //二分查找找到节点位置
    BT* node=bst;
    BT* father=NULL;
    while(node!=NULL && node->data!=key){
        father=node;
        if(key<node->data) node=node->left;
        else node=node->right;
    }
    if(node==NULL) return bst;
    //删除结点的左右子树非空，使其变为至多有有一个子节点的情况，找到后继节点替换
    if(node->left!=NULL && node->right!=NULL){
        BT* temp=node;
        father=node;
        node=node->right;
        while(node->left!=NULL){
            father=node;
            node=node->left;
        }
        temp->data=node->data;
    }
    //删除
    BT*node_ptr=NULL; //把子树先存起来
    if(node->left!=NULL) node_ptr=node->right;
    else if(node->right!=NULL) node_ptr=node->left;
```

```
        if(father==NULL) bst=node_ptr;
        else if(node==father->left) father->left=node_ptr;
        else father->right=node_ptr;
    }
```

主函数2

```
int main(){
    int a=0;
    BT*tree=new node(a);
    return 0;
}
```

AVL树😵

AVL头文件

```
#include<iostream>
#include<vector>
#include<string>
#include<algorithm>
using namespace std;
```

构建二叉树结构体3

```
//构建二叉树结构体
typedef struct node{
    int data;
    int height;  //注意新加进结构体的属性
    node*left,*right;
    node(int x):data(x),left(NULL),right(NULL),height(1){
    }  //构造函数
}BT;
```

//获取高度

时间复杂度O(1)

```
//获取高度
int GetHeight(BT*tree){
```

```
    if(tree=NULL) return 0;
    else return tree->height;
}
```

更新高度

时间复杂度O(1)

```
//更新高度
//想法：在插入一个节点需要更新之前所有节点，在有维护函数后感觉不需要了
void UpdateHeight(BT*tree){
    if(tree!=NULL){
        int height_l=GetHeight(tree->left);
        int height_r=GetHeight(tree->right);
        tree->height=max(height_l,height_r)+1;
    }
}
```

维护高度属性

时间复杂度O(n)

```
//维护高度属性，返回树的高度
int MaintainHeight(BT*tree){
    if(tree->left==NULL && tree->right==NULL) return 1;
    int height_l=MaintainHeight(tree->left);
    int height_r=MaintainHeight(tree->right);
    tree->height=max(height_l,height_r)+1;
    return tree->height;
}
```

左右旋

时间复杂度O(1)

```
//左右旋
//先更新子树再更新树
BT* LeftRotate(BT*tree){
    BT*node=tree->right;
    tree->right=node->left;
    UpdateHeight(tree);
    node->left=tree;
    UpdateHeight(node);
    return node;
}
```

```
BT* RightRotate(BT*tree){
    BT*node=tree->left;
    tree->left=node->right;
    UpdateHeight(tree);
    node->right=tree;
    UpdateHeight(node);
    return node;
}
```

插入2

```
//插入
//与二叉平衡树相同
BT* Insert(BT* bst, int key){
    if(bst==NULL) bst=new node(key);
    else if(key<bst->data) bst->left=Insert(bst->left,key);
    else bst->right=Insert(bst->right,key);
    return bst;
}
```

删除2

```
//删除
//与二叉平衡树相同
BT* Removal(BT*bst, int key){
    //二分查找找到节点位置
    BT* node=bst;
    BT* father=NULL;
    while(node!=NULL && node->data!=key){
        father=node;
        if(key<node->data) node=node->left;
        else node=node->right;
    }
    if(node==NULL) return bst;
    //删除结点的左右子树非空，使其变为至多有有一个子节点的情况，找到后继节点替换
    if(node->left!=NULL && node->right!=NULL){
        BT* temp=node;
        father=node;
        node=node->right;
        while(node->left!=NULL){
            father=node;
            node=node->left;
        }
        temp->data=node->data;
    }
    //删除
    BT*node_ptr=NULL;
    if(node->left!=NULL) node_ptr=node->right;
    else if(node->right!=NULL) node_ptr=node->left;
```

```
    if(father==NULL) bst=node_ptr;
    else if(node==father->left) father->left=node_ptr;
    else father->right=node_ptr;
    return bst;
}
```

自平衡😨

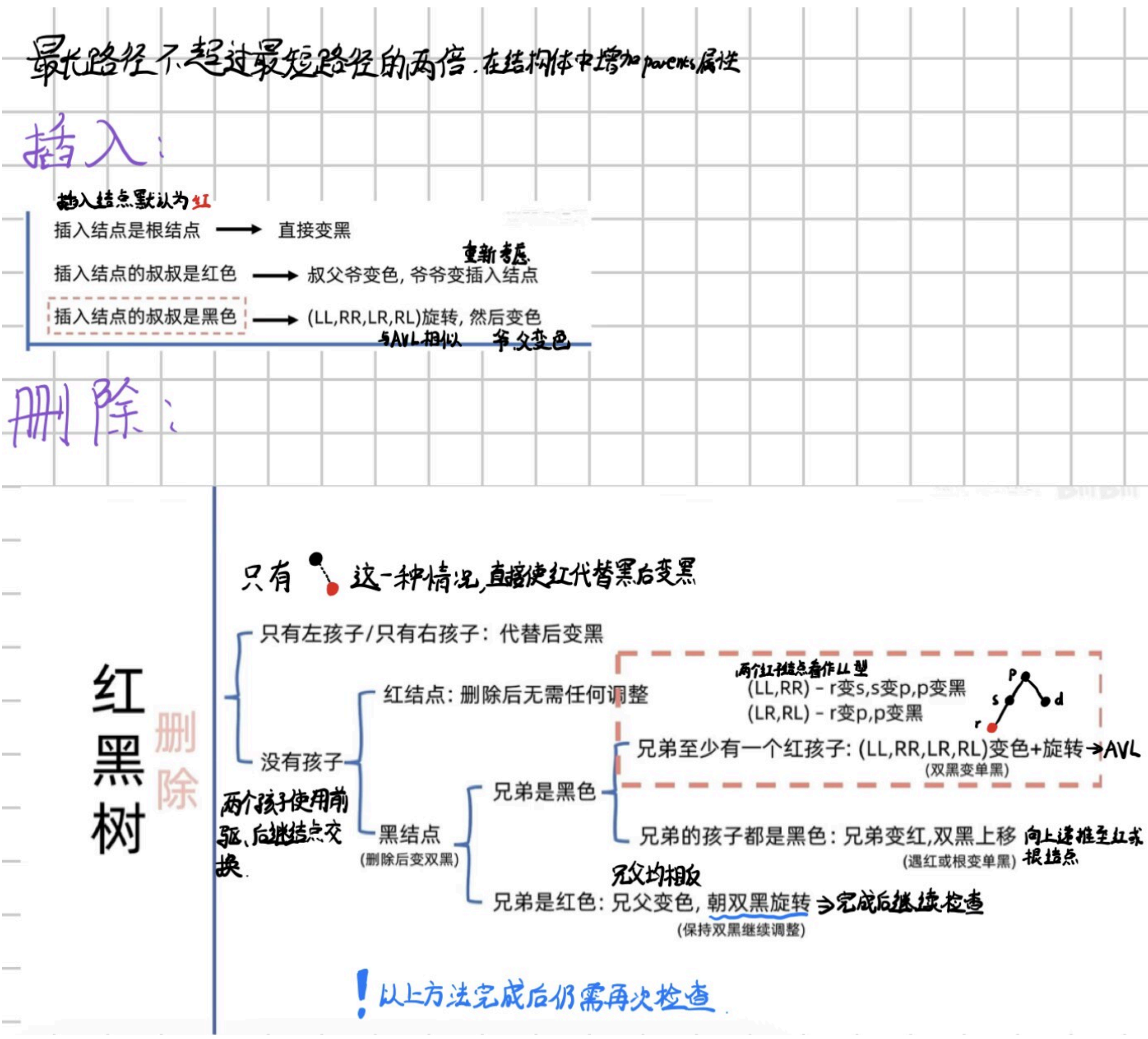
时间复杂度O(log(n))

```
//自平衡,插入删除可使用上述函数，node表示插入节点或删除结点的父节点
BT* Balance(BT*tree,BT*node){
    if(tree!=node){
        if(tree->data<node->data){
            tree->left=Balance(tree->left,node);
        }
        else{
            tree->right=Balance(tree->right,node);
        }   //保存根到node的路径
    }   //某一子树tree=node
    UpdateHeight(tree);

    if(GetHeight(tree->left)-GetHeight(tree->right)==2){
        if(GetHeight(tree->left)<GetHeight(tree->right)){
            tree->left=LeftRotate(tree->left);
        }
        tree=RightRotate(tree);
    }
    if(GetHeight(tree->right)-GetHeight(tree->left)==2){
        if(GetHeight(tree->right)<GetHeight(tree->left)){
            tree->right=RightRotate(tree->left);
        }
        tree=LeftRotate(tree);
    }
    return tree;
}
```

应用:树高上界

红黑树



二叉堆BinaryHeap😜

最大堆,最小堆,最大最小堆,对顶堆(找到第k小的元素)

头文件及定义

```
#include<iostream>
#include<vector>
#include<string>
using namespace std;

vector<int>h;   //未赋值
```

上调

时间复杂度O(log(n))

```
//上调
void SiftUp(vector<int> h,int i){  //i是起始位置
    int elem=h[i];
    while(i>1 && elem<h[i/2]){
        h[i]=h[i/2];
        i=i/2;
    }
    h[i]=elem;
}
```

下调

时间复杂度O(log(n))

```
//下调
void SiftDown(vector<int> h,int i){
    int last=h.size();
    int elem=h[i];
    int child=2*i;
    while(true){
        //找到更小的子节点
        if(child<last && h[child]>h[child+1]){
            child+=1;
        }
        else if(child>last){
            break;
        }
        //下调
        if(h[child]<elem){
            h[i]=h[child];
            i=child;
        }
        else{
            break;
        }
    }
    h[child]=elem;
}
```

插入

时间复杂度O(log(n))

```
//插入
void insert(vector<int>h,int x){
    h.push_back(x);
    SiftUp(h,h.size());
}
```

删除顶元素

时间复杂度O(log(n))

```
//删除顶元素
int DeleteMin(vector<int>h){
    int min=h[0];
    h[0]=h[h.size()];
    SiftDown(h,0);
    return min;
}
```

朴素建堆

时间复杂度O(n*log(n))

```
//朴素建堆
void MakeHeap(vector<int>h){
    for(int i=1;i<h.size();i++){
        SiftUp(h,i);
    }
}
```

快速建堆

时间复杂度O(n)

```
//快速建堆
void MakeHeapDown(vector<int>h){
    for(int i=(h.size()/2);i>=1;i--){
        SiftDown(h,i);
```

```
    }
}
```

静态查找Search 😵‍💫

顺序查找 🤪

查找最大最小值

时间复杂度3/2*n

```
#include<iostream>
using namespace std;
# define n 100

int main(){
    int a[n];
    int max=a[0],min=a[0];
    int k=n%2;
    while(k<n-1){
        if(a[k]<a[k+1]){
            if(min>a[k]){
                min=a[k];
            }
            if(max<a[k+1]){
                max=a[k+1];
            }
        }
        else{
            if(min>a[k+1]){
                min=a[k+1];
            }
            if(max<a[k]){
                max=a[k];
            }
        }
        k+=2;
    }
    return 0;
}
```

查找素数

埃氏筛选法 🤪

时间复杂度O(n*log(log(n)))

```
vector<int> ESearch(int n){
    vector<int>v;
    bool prime[n+1];
    prime[0]=false;
    prime[1]=false;
    for(int i=2;i<=n;i++){
        prime[i]=true;
    }
    for(int i=2;i<=n;i++){
        if(prime[i]==true){
            v.push_back(i);
            int m=2*i;
            while(m<=n){
                prime[m]=false;
                m+=i;
            }
        }
    }
}
```

欧拉筛选法😜

时间复杂度O(n)

```
vector<int> EulerSearch(int n){
    vector<int>v;
    bool prime[n+1];
    prime[0]=false;
    prime[1]=false;
    for(int i=2;i<=n;i++){
        prime[i]=true;
    }
    for(int i=2;i<=n;i++){
        if(prime[i]==true){
            v.push_back(i);
        }
        int k=0;
        while(i*v[k]<=n){
            prime[i*v[k]]=false;
            if(i%v[k]==0){
                break;
            }
            else{
                k++;
            }
        }
    }
}
```

二分查找😜

二分的思想十分重要

查找顺序表元素

时间复杂度O(log(n))

```
int BinarySearch(int a[],int left,int right,int key){
    int low=left-1;
    int high=right+1;
    while(high-low==1){
        int mid=(high+low)/2;
        if(a[mid]=key){
            return mid;
        }
        else if(a[mid]<key){
            high=mid;
        }
        else{
            low=mid;
        }
    }
    return -1;
}
```

未排序序列快速查找k小元素

先进行二分排序，再进行二分查找 时间复杂度O(1)--O(n^2^)

寻找最大的最小距离

时间复杂度O(n*log(X[n]-X[1]))

```
//m头牛安放进n间牛舍中，牛舍间距不同,寻找最大的最小间距
//转化为了找满足条件的最小(大)值
int MaxSpace(int a[],int n,int m){
    //a为n间牛舍的位置序列
    int min=0;
    int max=a[n-1]-a[0]+1;
    while(max-min>1){
        int mid=(max+min)/2;
        int num=1; //第num头牛
        int pre=0; //第num头牛在的位置
        for(int k=1;k<n;k++){
            if(a[k]-a[pre]>=mid){
                pre=k;
```

```
                num++;
            }
        }
        if(num==m){
            return mid;
        }
        else if(num>m){
            min=mid;
        }
        else{
            max=mid;
        }
    }
}
```

图graph 😵

图基础功能^1

头文件

```
#include<iostream>
#include<vector>
#include<queue>
using namespace std;
# define maxnum 100
```

邻接矩阵的存储结构

```
struct Graph1{
    int Vex[maxnum]; //顶点表---int表示符号位置
    int Edge[maxnum][maxnum]; //边表---含权值
    int Vnum1,Enum1; //点，边数
};
```

邻接链表表示

其实用一个二重vector也能实现链表操作，不要痴迷于链表

```
struct Edge{  //相邻点链表
    int src;      //表示该节点位置
    int weight;  //权重
    Edge *next;  //指向下一节点
    Edge(int x,int y):src(x),weight(y),next(NULL){}
};

struct Vex{
    int data;
    Edge *adj; //指向相邻点链表
};

struct Graph2{
    Vex vex[maxnum];
    int Vnum2,Enum2;
};
Edge* edge = new Edge(a); //构造图中线
```

合理选择存储方式

用邻接矩阵法删除节点

```
Graph1 RemoveVex1(Graph1 graph,int v){
    if(v<0 &&v>graph.Vnum1){
        return graph;
    }
    //计算该节点所连边的数量
    int count=0;
    for(int u=0;u<graph.Vnum1;u++){   //删除射出的边
        if(graph.Edge[v][u]) count++;
    }
    for(int u=0;u<graph.Vnum1;u++){   //删除射入的边
        if(graph.Edge[u][v]) count++;
    }
    //改变图信息
    graph.Vex[v]=graph.Vex[graph.Vnum1-1];   //替换节点信息

    for(int u=0;u<graph.Enum1;u++){   //改变边邻接矩阵的行
        graph.Edge[v][u]=graph.Edge[graph.Vnum1][u];
    }

    for(int u=0;u<graph.Enum1;u++){   //改变边邻接矩阵的列
        graph.Edge[u][v]=graph.Edge[u][graph.Vnum1];
    }

    graph.Enum1-=count;
    graph.Vnum1-=1;

    return graph;
}
```

深度优先遍历

时间复杂度O(V+E)

DFS算法实现

```
#include <iostream>
#include <vector>
using namespace std;

// 图的邻接表表示
struct Graph {
    int V; // 顶点数
    vector<vector<int>> adj; // 邻接表

    Graph(int V) : V(V), adj(V) {}

    void addEdge(int u, int v) {
        adj[u].push_back(v); // 添加边 u -> v
        adj[v].push_back(u); // 添加边 v -> u（如果是无向图）
    }
};

// DFS 递归核心代码
void DFSUtil(const Graph& graph, int node, vector<bool>& visited) {
    visited[node] = true; // 标记当前节点已访问
    cout << node << " ";  // 输出当前节点

    for (int neighbor : graph.adj[node]) { // 遍历邻接节点
        if (!visited[neighbor]) {           // 如果未访问
            DFSUtil(graph, neighbor, visited);
        }
    }
}

// 外部调用 DFS
void DFS(const Graph& graph, int start) {
    vector<bool> visited(graph.V, false); // 访问标记
    DFSUtil(graph, start, visited);
}

int main() {
    Graph graph(5); // 创建包含5个节点的图
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 4);

    cout << "DFS starting from node 0: ";
    DFS(graph, 0); // 从节点 0 开始 DFS
    cout << endl;
```

```
    return 0;
}
```

应用：寻找路径

给定图，判断两点之间是否存在路径

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

// 图的邻接表表示
struct Graph {
    int V; // 顶点数
    vector<vector<int>> adj; // 邻接表

    Graph(int V) : V(V), adj(V) {}

    void addEdge(int u, int v) {
        adj[u].push_back(v); // 添加边 u -> v
    }
};

// 深度优先搜索寻找路径
bool DFS_Find(const Graph& graph, int v, int t, vector<bool>& visited, vector<int>&
pre) {
    visited[v] = true; // 标记当前节点已访问
    if (v == t) return true; // 如果找到目标节点，返回 true

    for (int neighbor : graph.adj[v]) { // 遍历邻接节点
        if (!visited[neighbor]) { // 如果未访问
            pre[neighbor] = v; // 记录前驱节点
            if (DFS_Find(graph, neighbor, t, visited, pre)) {
                return true; // 如果找到路径，直接返回 true
            }
        }
    }
    return false; // 未找到路径
}

// 寻找从 s 到 t 的路径
void FindPath(const Graph& graph, int s, int t) {
    vector<bool> visited(graph.V, false); // 访问标记
    vector<int> pre(graph.V, -1); // 记录路径

    if (DFS_Find(graph, s, t, visited, pre)) { // 如果找到路径
        stack<int> path;
        for (int v = t; v != -1; v = pre[v]) {
            path.push(v); // 逆向存储路径
        }

        // 输出路径
```

```
        while (!path.empty()) {
            cout << path.top();
            path.pop();
            if (!path.empty()) cout << " -> ";
        }
        cout << endl;
    } else {
        cout << "No Path" << endl;
    }
}

// 主函数示例
int main() {
    Graph graph(5); // 创建包含 5 个节点的图
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 4);
    graph.addEdge(3, 4);

    int s = 0, t = 4; // 起点和终点
    cout << "Finding path from " << s << " to " << t << ":\n";
    FindPath(graph, s, t);

    return 0;
}
```

应用：走迷宫

```
bool Find_PathMaze(int *map[],int m,int n,int x,int y,int tx,int ty,int *pre[]){
    bool visit[m][n];
    for(int i=0;i<m;i++){
        for(int j=0;j<n;j++){
            visit[i][j]=false;
        }
    }
    visit[x][y]=true;

    int adj[4][2]={{-1,0},{1,0},{0,-1},{0,1}};

    if(x==tx && y==ty){
        return true;
    }
    for(int k=0;k<4;k++){
        int nx=x+adj[k][0];
        int ny=y+adj[k][1];
        if(nx>=0 && nx<m && ny>=0 && ny<n && map[nx][ny]!=1 && visit[nx]
[ny]==false){
            pre[nx][ny]=k;
            if(Find_PathMaze(map,m,n,nx,ny,tx,ty,pre)==true){
                return true;
            }
        }
    }
```

```
    return false;
}
```

广度优先遍历

时间复杂度O(V+E)

BFS算法实现

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

// 图的邻接表表示
struct Graph {
    int V; // 顶点数
    vector<vector<int>> adj; // 邻接表

    Graph(int V) : V(V), adj(V) {}

    void addEdge(int u, int v) {
        adj[u].push_back(v); // 添加边 u -> v
        adj[v].push_back(u); // 添加边 v -> u（如果是无向图）
    }
};

// BFS 核心代码
void BFS(const Graph& graph, int start) {
    vector<bool> visited(graph.V, false); // 访问标记
    queue<int> q; // 队列

    q.push(start); // 起始点入队
    visited[start] = true; // 标记起始点已访问

    while (!q.empty()) {
        int node = q.front(); // 取队首元素
        q.pop();
        cout << node << " "; // 输出当前节点

        for (int neighbor : graph.adj[node]) { // 遍历邻接节点
            if (!visited[neighbor]) {          // 如果未访问
                visited[neighbor] = true;      // 标记为已访问
                q.push(neighbor);              // 入队
            }
        }
    }
}

int main() {
    Graph graph(5); // 创建包含5个节点的图
```

```
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 4);

    cout << "BFS starting from node 0: ";
    BFS(graph, 0); // 从节点 0 开始 BFS
    cout << endl;

    return 0;
}
```

例题

跳一跳，只有三个位置，求到达终点的最少次数

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
using namespace std;

int minJumpsToReachEnd(const vector<int>& nums) {
    int n = nums.size();
    if (n == 1) return 0; // 如果数组长度为1，已经在末尾，不需要跳跃

    // BFS 初始化，BFS思想
    queue<pair<int, int>> q;  // (index, jump_count)
    vector<bool> visited(n, false);
    unordered_map<int, vector<int>> value_map; // 存储值到索引的映射,即所有能跳到的地方

    // 将每个值的索引存储在 value_map 中
    for (int i = 0; i < n; ++i) {
        value_map[nums[i]].push_back(i);
    }

    // 初始状态：从0开始
    q.push({0, 0});
    visited[0] = true;

    // BFS
    while (!q.empty()) {
        auto [index, jumps] = q.front();
        q.pop();

        // 如果当前下标已经是最后一个位置，返回跳跃次数
        if (index == n - 1) {
            return jumps;
        }

        // 尝试跳到 i + 1
```

```
        if (index + 1 < n && !visited[index + 1]) {
            visited[index + 1] = true;
            q.push({index + 1, jumps + 1});  //注意每次加1
        }

        // 尝试跳到 i - 1
        if (index - 1 >= 0 && !visited[index - 1]) {
            visited[index - 1] = true;
            q.push({index - 1, jumps + 1});
        }

        // 尝试跳到与 nums[i] 相等的所有位置
        if (value_map.find(nums[index]) != value_map.end()) {
            // 访问当前值的所有相同值位置
            for (int next_index : value_map[nums[index]]) {
                if (!visited[next_index]) {
                    visited[next_index] = true;
                    q.push({next_index, jumps + 1});
                }
            }
            // 访问完后清空该值的所有位置，避免重复访问
            value_map.erase(nums[index]);
        }
    }

    return -1; // 如果无法到达最后一个位置，返回 -1
}

int main() {
    int n;
    cin >> n;
    vector<int> nums(n);
    for (int i = 0; i < n; ++i) {
        cin >> nums[i];
    }

    int result = minJumpsToReachEnd(nums);
    cout << result << endl;

    return 0;
}
```

欧拉回路🤯

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

// 检查图是否具有欧拉回路的函数
bool hasEulerianCircuit(const vector<vector<int>>& graph) {
```

```cpp
    int oddDegreeCount = 0;
    for (const auto& adj : graph) {
        if (adj.size() % 2 != 0) {
            oddDegreeCount++;
        }
    }
    return oddDegreeCount == 0;
}

// 使用Fleury算法查找欧拉回路的函数
void findEulerianCircuit(vector<vector<int>>& graph, int start) {
    stack<int> path;      // 用于存储当前路径的栈
    vector<int> circuit;  // 用于存储欧拉回路的向量

    path.push(start);

    while (!path.empty()) {
        int u = path.top();

        //如果当前顶点没有更多边
        if (graph[u].empty()) {
            circuit.push_back(u);
            path.pop();
        } else {
            // 移动到下一个顶点并删除边缘
            int v = graph[u].back();
            graph[u].pop_back();
            // 从 v 到 u 删除反向边
            auto it = find(graph[v].begin(), graph[v].end(), u);
            if (it != graph[v].end()) {
                graph[v].erase(it);
            }
            path.push(v);
        }
    }

    // 打印欧拉回路
    cout << "Eulerian Circuit: ";
    for (int v : circuit) {
        cout << v << " ";
    }
    cout << endl;
}

int main() {
    int n, m;
    cout << "Enter the number of vertices and edges: ";
    cin >> n >> m;

    vector<vector<int>> graph(n);

    cout << "Enter the edges (u v):\n";
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        graph[u].push_back(v);
        graph[v].push_back(u);  // 图是无向的
```

```
    }

    if (!hasEulerianCircuit(graph)) {
        cout << "The graph does not have an Eulerian circuit." << endl;
    } else {
        int start = 0;  // 起始顶点（可以是任何有边的顶点）
        findEulerianCircuit(graph, start);
    }

    return 0;
}
```

无向图的割点与桥🤯

时间复杂度O(V+E)

```
//DFS求图的割点(Tarjan算法)
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

const int MAXN = 10000; // 最大顶点数量
vector<int> adj[MAXN]; // 邻接表表示图
vector<bool> visited(MAXN, false); // 记录是否访问过
vector<int> discovery(MAXN, -1); // 记录发现时间
vector<int> low(MAXN, -1); // 记录子树中最早被访问的顶点时间
vector<int> parent(MAXN, -1); // 记录顶点的父节点
vector<int> articulationPoints; // 存储割点
vector<pair<int, int>> bridges; // 存储桥

int timeCounter = 0; // 时间计数器

void DFS(int u) {
    visited[u] = true;
    discovery[u] = low[u] = ++timeCounter;
    int children = 0;

    for (int v : adj[u]) {
        if (!visited[v]) {
            children++;
            parent[v] = u;
            DFS(v);

            // 更新 low[u]，以包含子树中的最小值
            low[u] = min(low[u], low[v]);

            // 判断割点
            if (parent[u] == -1 && children > 1) {
                articulationPoints.push_back(u);
            }
            if (parent[u] != -1 && low[v] >= discovery[u]) {
```

```
                articulationPoints.push_back(u);
            }

            // 判断桥
            if (low[v] > discovery[u]) {
                bridges.push_back({u, v});
            }
        } else if (v != parent[u]) {
            // 更新 low 值，处理回边
            low[u] = min(low[u], discovery[v]);
        }
    }
}

void findArticulationPointsAndBridges(int n) {
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            DFS(i);
        }
    }
}

int main() {
    int n, m;
    cout << "Enter number of vertices and edges: ";
    cin >> n >> m;

    cout << "Enter the edges (u v):\n";
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u); // 无向图
    }

    findArticulationPointsAndBridges(n);

    cout << "Articulation Points:\n";
    for (int point : articulationPoints) {
        cout << point << " ";
    }
    cout << endl;

    cout << "Bridges:\n";
    for (auto bridge : bridges) {
        cout << bridge.first << " - " << bridge.second << endl;
    }

    return 0;
}
//对于树边(u,v)如果dfn(u)<low(v),则(u,v)是桥
```

有向图的强连通分量🤯

时间复杂度O(V+E)

kosaraju算法😫

```
#include <iostream>
#include <vector>
#include <stack>
#include <algorithm>
using namespace std;

const int MAXN = 10000; // 最大顶点数量
vector<int> adj[MAXN];  // 原图的邻接表
vector<int> revAdj[MAXN]; // 反转图的邻接表
vector<bool> visited(MAXN, false); // 记录是否访问过
stack<int> finishStack; // 存储第一次 DFS 的完成顺序
vector<vector<int>> scc; // 存储所有强连通分量

// 第一次 DFS：记录顶点的完成顺序
void dfs1(int u) {
    visited[u] = true;
    for (int v : adj[u]) {
        if (!visited[v]) {
            dfs1(v);
        }
    }
    finishStack.push(u);
}

// 第二次 DFS：在反转图中找到一个强连通分量
void dfs2(int u, vector<int>& component) {
    visited[u] = true;
    component.push_back(u);
    for (int v : revAdj[u]) {
        if (!visited[v]) {
            dfs2(v, component);
        }
    }
}

// Kosaraju 主函数
void kosarajuSCC(int n) {
    // 第一次 DFS：记录完成顺序
    fill(visited.begin(), visited.begin() + n, false);
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            dfs1(i);
        }
    }

    // 反转图
    for (int u = 0; u < n; u++) {
        for (int v : adj[u]) {
            revAdj[v].push_back(u);
        }
```

```
    }

    // 第二次 DFS: 按完成顺序处理反转图
    fill(visited.begin(), visited.begin() + n, false);
    while (!finishStack.empty()) {
        int u = finishStack.top();
        finishStack.pop();
        if (!visited[u]) {
            vector<int> component;
            dfs2(u, component);
            scc.push_back(component);
        }
    }
}

int main() {
    int n, m;
    cout << "Enter number of vertices and edges: ";
    cin >> n >> m;

    cout << "Enter the edges (u v):\n";
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v); // 有向图
    }

    kosarajuSCC(n);

    cout << "Strongly Connected Components:\n";
    for (const auto& component : scc) {
        for (int v : component) {
            cout << v << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Tarjan算法😫

```
#include <iostream>
#include <vector>
#include <stack>
#include <algorithm>
using namespace std;

const int MAXN = 10000; // 最大顶点数量
vector<int> adj[MAXN];  // 邻接表表示图
vector<int> discovery(MAXN, -1); // 记录发现时间
vector<int> low(MAXN, -1); // 记录子树中最早被访问的顶点时间
vector<bool> onStack(MAXN, false); // 判断顶点是否在栈中
```

```cpp
stack<int> stk;  // 栈，用于追踪当前访问路径
vector<vector<int>> scc; // 存储所有强连通分量

int timeCounter = 0; // 时间计数器

void tarjanDFS(int u) {
    discovery[u] = low[u] = ++timeCounter;
    stk.push(u);
    onStack[u] = true;

    for (int v : adj[u]) {
        if (discovery[v] == -1) {
            // v 未访问
            tarjanDFS(v);
            low[u] = min(low[u], low[v]);
        } else if (onStack[v]) {
            // v 在栈中，是一条回边
            low[u] = min(low[u], discovery[v]);
        }
    }

    // 如果 u 是一个强连通分量的根
    if (low[u] == discovery[u]) {
        vector<int> component;
        while (true) {
            int v = stk.top();
            stk.pop();
            onStack[v] = false;
            component.push_back(v);
            if (v == u) break;
        }
        scc.push_back(component);
    }
}

void findSCCs(int n) {
    for (int i = 0; i < n; i++) {
        if (discovery[i] == -1) {
            tarjanDFS(i);
        }
    }
}

int main() {
    int n, m;
    cout << "Enter number of vertices and edges: ";
    cin >> n >> m;

    cout << "Enter the edges (u v):\n";
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v); // 有向图
    }

    findSCCs(n);
```

```
    cout << "Strongly Connected Components:\n";
    for (const auto& component : scc) {
        for (int v : component) {
            cout << v << " ";
        }
        cout << endl;
    }

    return 0;
}
```

拓扑排序🤯

课本伪代码重写

时间复杂度O(V+E)

```
//BFS算法(顺序为BFS近似的顺序)
void CountInDegree(Graph2 graph,int n,int indegree[]){
    int indegree[n];
    for(int i=0;i<n;i++){
        indegree[i]=0;
    }
    for(int i=0;i<n;i++){
        Edge* p=graph.vex[i].adj;
        while(p!=NULL){
            indegree[p->src]++;
            p=p->next;
        }
    }
}

void TopSort_BFS(Graph2 graph,int n){
    vector<int>top_list;
    int indegree[n];  //入度
    CountInDegree(graph,n,indegree);
    queue<int>queue;
    for(int i=0;i<n;i++){
        if(indegree[i]==0){
            queue.push(i);
        }
    }
    while(!queue.empty()){
        int u=queue.front();
        queue.pop();
        top_list.push_back(u);
        Edge* p=graph.vex[u].adj; //邻接节点入度减一
        while(p!=NULL){
            indegree[p->src]--;
            if(indegree[p->src]==0){
                queue.push(p->src);
```

```
            }
            p=p->next;
        }
    }
}
//DFS算法(顺序为先输出一条路，再看其他路)
void DFS_Sort(Graph2 graph,int v,bool visit[],vector<int>top_list){
    visit[v]=true;
    Edge* p=graph.vex[v].adj;
    while(p!=NULL){
        if(visit[p->src]==false){
            DFS_Sort(graph,p->src,visit,top_list);
        }
        p=p->next;
    }
    top_list.push_back(v);
}


void TopSort_DFS(Graph2 graph,int n,vector<int>top_list){
    bool visit[n];
    for(int i=0;i<n;i++){
        visit[i]=false;
    }
    for(int v=0;v<n;v++){
        if(visit[v]==false){
            DFS_Sort(graph,v,visit,top_list);
        }
    }
}
```

例题1 😫

```
#include <iostream>
#include <vector>
#include <queue>
#include <cstring>
using namespace std;

int main() {
    int n, m;
    while (cin >> n >> m) { // 读取顶点数 n 和边数 m，支持多组数据
        vector<vector<int>> adj(n + 1); // 邻接表表示图，1-based 索引
        vector<int> indegree(n + 1, 0); // 存储每个节点的入度
        vector<int> result;             // 存储拓扑排序结果

        // 读取边并建立图
        for (int i = 0; i < m; i++) {
            int d, u;
            cin >> d >> u;          // 从 d 指向 u 的边
            adj[d].push_back(u);    // d 的邻接点增加 u
            indegree[u]++;          // u 的入度增加
        }
```

```
        queue<int> q; // 队列，用于执行拓扑排序
        for (int i = 1; i <= n; i++) { // 将所有入度为 0 的节点加入队列
            if (indegree[i] == 0) {
                q.push(i);
            }
        }

        int count = 0; // 记录处理过的节点数量
        bool unique = true; // 标志是否存在唯一的拓扑排序

        // 开始拓扑排序
        while (!q.empty()) {
            if (q.size() > 1) { // 如果队列中有多个节点，则拓扑排序不唯一
                unique = false;
            }
            int node = q.front(); // 取出队首节点
            q.pop();
            result.push_back(node); // 将节点加入结果中
            count++; // 计数已排序的节点数

            for (int next : adj[node]) { // 遍历当前节点的所有邻接点
                if (--indegree[next] == 0) { // 入度减 1，如果变为 0，则加入队列
                    q.push(next);
                }
            }
        }

        // 判断拓扑排序结果
        if (count != n) { // 如果没有处理完所有节点，则图中存在环
            cout << "0" << endl; // 图中有环，无拓扑排序
        } else {
            if (unique) {
                cout << "1" << endl; // 图有唯一的拓扑排序
            } else {
                cout << "2" << endl; // 图有多个可能的拓扑排序
            }
        }
    }
    return 0;
}
```

最短路径🤯

Dijkstra算法(边权重非负)*😫

时间复杂度O(E+VlogV)

```
#include <iostream>
#include <vector>
#include <queue>
```

```cpp
#include <climits>
using namespace std;

const int INF = INT_MAX; // 定义无穷大

// Dijkstra 核心函数
void Dijkstra(int start, int n, const vector<vector<pair<int, int>>>& graph,
vector<int>& dist) {
    dist.assign(n, INF);            // 初始化所有点的距离为无穷大
    dist[start] = 0;                // 起点距离自身为0
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
    pq.emplace(0, start);           // 优先队列存储 (距离, 节点)

    while (!pq.empty()) {
        auto [d, u] = pq.top();     // 获取当前距离最小的节点
        pq.pop();

        if (d > dist[u]) continue;  // 如果队列中的距离已经过时，跳过

        // 遍历节点 u 的所有邻居
        for (const auto& [v, weight] : graph[u]) {
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.emplace(dist[v], v); // 更新优先队列
            }
        }
    }
}

int main() {
    int n, m, start;
    cin >> n >> m >> start; // 输入点数、边数、起点
    start--;                // 转换为 0-based 索引

    vector<vector<pair<int, int>>> graph(n); // 邻接表表示图
    for (int i = 0; i < m; ++i) {
        int u, v, w;
        cin >> u >> v >> w;
        u--, v--; // 转换为 0-based 索引
        graph[u].emplace_back(v, w);
        graph[v].emplace_back(u, w); // 若是无向图，需添加反向边
    }

    vector<int> dist; // 距离数组
    Dijkstra(start, n, graph, dist);

    // 输出结果
    for (int i = 0; i < n; ++i) {
        if (dist[i] == INF)
            cout << "INF ";
        else
            cout << dist[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Bellman-Ford算法😫

时间复杂度O(V^3^) [最坏情况]

```
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

const int INF = INT_MAX; // 定义无穷大

// Bellman-Ford 核心函数
bool BellmanFord(int start, int n, vector<tuple<int, int, int>>& edges,
vector<int>& dist) {
    dist.assign(n, INF); // 初始化所有点的距离为无穷大
    dist[start] = 0;     // 起点距离自身为0

    // 松弛操作，最多进行 n-1 次
    for (int i = 0; i < n - 1; ++i) {
        for (const auto& [u, v, weight] : edges) {
            if (dist[u] != INF && dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
            }
        }
    }

    // 检测负权环：如果还能松弛，说明存在负权环
    for (const auto& [u, v, weight] : edges) {
        if (dist[u] != INF && dist[u] + weight < dist[v]) {
            return false; // 发现负权环
        }
    }

    return true; // 没有负权环
}

int main() {
    int n, m, start;
    cin >> n >> m >> start; // 输入点数、边数、起点
    start--;                // 转换为 0-based 索引

    vector<tuple<int, int, int>> edges; // 存储边的三元组 (u, v, weight)
    for (int i = 0; i < m; ++i) {
        int u, v, w;
        cin >> u >> v >> w;
        u--, v--; // 转换为 0-based 索引
        edges.emplace_back(u, v, w);
    }

    vector<int> dist; // 距离数组
    if (BellmanFord(start, n, edges, dist)) {
        // 输出结果
        for (int i = 0; i < n; ++i) {
```

```
            if (dist[i] == INF)
                cout << "INF ";
            else
                cout << dist[i] << " ";
        }
        cout << endl;
    } else {
        cout << "Negative weight cycle detected." << endl;
    }

    return 0;
}
```

SPFA算法😫

时间复杂度O(V*E) [最坏情况]

空间复杂度O(V)

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

const int INF = INT_MAX; // 定义无穷大

// SPFA 核心函数
void SPFA(int start, int n, vector<vector<pair<int, int>>>& graph) {
    vector<int> dist(n, INF);       // 距离数组，初始为无穷大
    vector<bool> inQueue(n, false); // 标记节点是否在队列中
    queue<int> q;                   // 辅助队列

    dist[start] = 0; // 起点到自身的距离为0
    q.push(start);
    inQueue[start] = true;

    while (!q.empty()) {
        int u = q.front();
        q.pop();
        inQueue[u] = false;

        // 遍历 u 的所有邻接边
        for (const auto& [v, weight] : graph[u]) {
            if (dist[u] + weight < dist[v]) { // 松弛操作
                dist[v] = dist[u] + weight;
                if (!inQueue[v]) { // 如果 v 不在队列中，加入队列
                    q.push(v);
                    inQueue[v] = true;
                }
            }
        }
    }
```

```
    // 输出结果
    for (int i = 0; i < n; ++i) {
        if (dist[i] == INF)
            cout << "INF ";
        else
            cout << dist[i] << " ";
    }
    cout << endl;
}

int main() {
    int n, m, start;
    cin >> n >> m >> start; // 输入点数、边数、起点
    start--;                // 转换为0-based

    // 构建邻接表
    vector<vector<pair<int, int>>> graph(n);
    for (int i = 0; i < m; ++i) {
        int u, v, w;
        cin >> u >> v >> w;
        u--, v--; // 转换为0-based
        graph[u].emplace_back(v, w);
    }

    // 调用 SPFA
    SPFA(start, n, graph);

    return 0;
}
```

最小生成树🤯

求解权值最小的生成树

Prim算法(扩点)😫

与Dijkstra算法完全相同，只有对临界节点的判断不同

时间复杂度O(V+ElogE)

空间复杂度O(V+E)

prim算法实现[3]

```
#include <iostream>
#include <vector>
```

```cpp
#include <queue>
#include <tuple>
#include <algorithm>
using namespace std;

// 定义边的结构
struct Edge {
    int to, weight;
    Edge(int t, int w) : to(t), weight(w) {}
};

// 最小生成树 Prim 实现
int main() {
    int n, m, start;
    cin >> n >> m >> start; // 输入村庄数、边数、起始编号
    start--; // 转换为 0-based 编号

    // 邻接表存储图
    vector<vector<Edge>> graph(n);
    for (int i = 0; i < m; ++i) {
        int u, v, w;
        cin >> u >> v >> w;
        u--; v--; // 转换为 0-based 编号
        graph[u].emplace_back(v, w);
        graph[v].emplace_back(u, w); // 因为是无向图
    }

    // 最小生成树的边
    vector<tuple<int, int, int>> mstEdges;

    // 优先队列存储 {边权重, 当前节点, 前驱节点}
    priority_queue<tuple<int, int, int>, vector<tuple<int, int, int>>, greater<>>
pq;
    vector<bool> inMST(n, false); // 标记节点是否已加入 MST

    // 初始化，从起始节点开始
    inMST[start] = true;
    for (const auto& edge : graph[start]) {
        pq.emplace(edge.weight, edge.to, start);
    }

    // 构造 MST
    while (!pq.empty()) {
        auto [weight, to, from] = pq.top();  //注意此处赋值，是中括号
        pq.pop();

        // 如果目标节点已在 MST 中，跳过
        if (inMST[to]) continue;

        // 加入 MST
        inMST[to] = true;
        mstEdges.emplace_back(from, to, weight);  //注意此处不需要大括号

        // 将目标节点的所有邻接边加入队列
        for (const auto& edge : graph[to]) {
            if (!inMST[edge.to]) {
                pq.emplace(edge.weight, edge.to, to);
```

```
            }
        }

        // 如果 MST 边数等于 n - 1，则完成
        if (mstEdges.size() == n - 1) break;
    }

    // 输出 MST 边
    for (auto& [from, to, weight] : mstEdges) {  //注意这里是中括号
        // 转换回 1-based 编号并输出
        if (from > to) swap(from, to);
        cout << from + 1 << "," << to + 1 << "," << weight << endl;
    }

    return 0;
}
```

Kruskal算法(扩边) 😫

并查集

类似于集合合并

```
// 并查集的结构体
class UnionFind {
public:
    vector<int> parent, rank;

    UnionFind(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    // 查找父节点并进行路径压缩
    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    // 合并两个集合
    void union_sets(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX != rootY) {
            // 按照秩（rank）优化合并
            if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
```

```
            } else if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
};
```

Kruskal算法实现^3

实现的很简单，不要想太多，而且不要将思维局限于创造图结构，图也是存在顺序表中的

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

// 边的结构体
struct Edge {
    int u, v, weight;
    Edge(int a, int b, int w) : u(a), v(b), weight(w) {}

    // 按照权重从小到大排序
    bool operator<(const Edge& other) const {
        return weight < other.weight;
    }
};

// 并查集的结构体
class UnionFind {
public:
    vector<int> parent, rank;

    UnionFind(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    // 查找父节点并进行路径压缩
    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    // 合并两个集合
    void union_sets(int x, int y) {
        int rootX = find(x);
```

```cpp
        int rootY = find(y);
        if (rootX != rootY) {
            // 按照秩（rank）优化合并
            if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
};

void kruskal(int n, vector<Edge>& edges) {
    UnionFind uf(n);
    vector<Edge> mst;

    // 对边按权重升序排序
    sort(edges.begin(), edges.end());

    for (const auto& edge : edges) {
        int u = edge.u, v = edge.v, w = edge.weight;
        if (uf.find(u) != uf.find(v)) {
            uf.union_sets(u, v);
            mst.push_back(edge);
        }
    }

    // 输出最小生成树的边
    for (const auto& edge : mst) {
        if (edge.u < edge.v) {
            cout << (edge.u+1) << "," << (edge.v+1) << "," << edge.weight << endl;
        } else {
            cout << (edge.v+1) << "," << (edge.u+1) << "," << edge.weight << endl;
        }
    }
}

int main() {
    int n, m;
    cin >> n >> m;

    vector<Edge> edges;
    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        edges.push_back(Edge(u - 1, v - 1, w));  // 转换为从0开始的索引
    }

    kruskal(n, edges);


    return 0;
}
```

排序优化(高度复杂)😫

```
struct Edge{  //相邻点链表
    int src;      //表示该节点位置
    int weight;  //节点间线的权重
    Edge *next;  //指向下一节点
    Edge(int x,int y):src(x),weight(y),next(NULL){}
};

struct Vex{
    int data;
    Edge *adj=NULL; //指向相邻点链表
};

struct Graph2{
    Vex vex[maxnum];
    int Vnum2,Enum2;
};

struct edge{
    int u;
    int v;
    int weight;
    edge(int x,int y,int z):u(x),v(y),weight(z){}
};


void SiftUp(vector<edge> h,int i){  //i是起始位置
    edge elem=h[i];
    while(i>1 && elem.weight<h[i/2].weight){
        h[i]=h[i/2];
        i=i/2;
    }
    h[i]=elem;
}

void SiftDown(vector<edge> h,int i){
    int last=h.size();
    edge elem=h[i];
    int child=2*i;
    while(true){
        //找到更小的子节点
        if(child<last && h[child].weight>h[child+1].weight){
            child+=1;
        }
        else if(child>last){
            break;
        }
        //下调
        if(h[child].weight<elem.weight){
            h[i]=h[child];
            i=child;
        }
```

```
        else{
            break;
        }
    }
    h[child]=elem;
}

void insert(vector<edge>h,edge x){
    h.push_back(x);
    SiftUp(h,h.size());
}

edge DeleteMin(vector<edge>h){
    edge min=h[0];
    h[0]=h[h.size()];
    SiftDown(h,0);
    return min;
}

//查找元素所在集合
int Find(int parent[],int a){
    int root=a;
    while(parent[root]!=root){
        root=parent[root];
    }
    //路径压缩，将a放在根下
    while(parent[a]!=root){
        int temp=parent[a];
        parent[a]=root;
        a=temp;
    }
    return root;
}

//合并两个元素所在集合
void Union(int parent[],int a,int b){
    int root_a=Find(parent,a);
    int root_b=Find(parent,b);
    if(root_a!=root_b){
        parent[root_b]=root_a;
    }
}

int Kruskal(Graph2 graph){
    vector<edge>v;  //堆
    int parent[graph.Vnum2];
    for(int i=0;i<graph.Vnum2;i++){
        parent[i]=i;
        Edge* p=graph.vex[i].adj;
        while(p!=NULL){
            edge e(i,p->src,p->weight);
            insert(v,e);
            p=p->next;
        }
    }

    int total_weight=0;
```

```
    while(!v.empty()){
        edge e=DeleteMin(v);
        if(Find(parent,e.u)!=Find(parent,e.v)){  //保证不会形成环
            total_weight+=e.weight;
            Union(parent,e.u,e.v);
        }
    }
    return total_weight;
}
```

图例题^2 🤯

例题一

求解图中所有点可到达点的总和

```
#include <iostream>
#include <bitset>
#include <vector>
using namespace std;

const int MAX_N = 2001;

char ch[MAX_N];
bitset<MAX_N> a[MAX_N]; // 使用 bitset 存储节点的连接关系
int ans;

int main() {
    int n;
    cin >> n;

    // 输入图的连接情况
    for (int i = 1; i <= n; i++) {
        cin >> (ch + 1); // 读取每一行的图的连接关系
        for (int j = 1; j <= n; j++) {
            if (ch[j] == '1') {
                a[i][j] = 1; // 如果有边，更新 bitset
            }
        }
        a[i][i] = 1; // 每个节点可以到达自己
    }

    // 使用 Floyd-Warshall 算法进行传递闭包计算
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            if (a[j][i]) {
                a[j] |= a[i]; // 如果 j 能到 i，就让 j 到达 i 能到的所有节点
                //上边这一步很重要，每行列的位或操作传递了连通性；
            }
        }
    }
```

```
    // 统计每个节点能到达的节点数
    for (int i = 1; i <= n; i++) {
        ans += a[i].count(); // 统计节点 i 能到达的节点数
    }

    cout << ans << endl; // 输出结果

    return 0;
}
```

例题二

并查集例题 所用想法

```
class Solution {
public:
    int findCircleNum(vector<vector<int>>& isConnected) {
        int n = isConnected.size();
        vector<int> root(n);

        for (int i = 0; i < n; ++i) {
            root[i] = i;
        }

        auto find = [&](int x) {
            while (x != root[x]) {
                root[x] = root[root[x]];
                x = root[x];
            }
            return x;
        };

        auto unite = [&](int x, int y) {
            int rootX = find(x);
            int rootY = find(y);
            if (rootX != rootY) {
                root[rootY] = rootX;
            }
        };

        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (isConnected[i][j] == 1) {
                    unite(i, j);
                }
            }
        }

        int count = 0;
        for (int i = 0; i < n; ++i) {
            if (find(i) == i) {    //很好的想法
```

```
                ++count;
            }
        }

        return count;
    }
};
```

例题三* 😫

[力扣原题（困难）]

> 真难理解啊!!! 😭

想成将分散的点从小到大连起来形成“好路径”， 而不是从最大开始查找删除

```
class Solution {
public:
    int numberOfGoodPaths(vector<int> &vals, vector<vector<int>> &edges) {
        int n = vals.size();
        vector<vector<int>> g(n);
        for (auto &e : edges) {
            int x = e[0], y = e[1];
            g[x].push_back(y);
            g[y].push_back(x); // 建图
        }

        // 并查集模板
        // size[x] 表示节点值等于 vals[x] 的节点个数，
        // 如果按照节点值从小到大合并，size[x] 也是连通块内的等于最大节点值的节点个数
        int id[n], fa[n], size[n]; // id 后面排序用
        iota(id, id + n, 0);
        iota(fa, fa + n, 0);
        fill(size, size + n, 1);
        function<int(int)> find = [&](int x) -> int { return fa[x] == x ? x : fa[x] = find(fa[x]); };

        int ans = n; // 单个节点的好路径
        sort(id, id + n, [&](int i, int j) { return vals[i] < vals[j]; });
        for (int x : id) {
            int vx = vals[x], fx = find(x);
            for (int y : g[x]) {  //对周围节点的情况分析
                y = find(y);
                if (y == fx || vals[y] > vx)
                    continue; // 只考虑最大节点值不超过 vx 的连通块
                if (vals[y] == vx) { // 可以构成好路径
                    ans += size[fx] * size[y]; // 乘法原理,两点之间只有唯一的路径，故有乘法原理
                    size[fx] += size[y]; // 统计连通块内节点值等于 vx 的节点个数
                }
                fa[y] = fx; // 把小的节点值合并到大的节点值上，相当于是周围节点的值小于x时的行为
```

```
            }
        }
        return ans;
    }
};
```

算法😵‍💫

动态规划，贪心算法，回溯算法，二分法,分治法，归并排序，DFS，BFS，递归，快速选择，记忆化搜索等等

数据结构学习列表🤡

- [ ] 栈与队列
- [ ] 排序算法
- [ ] 二叉树
- [ ] 二叉堆
- [ ] 静态查找
- [ ] 图
- [ ] 算法

==一点都没学呢吧🤡🤡🤡==