搜索

搜索，也就是对状态空间进行枚举，通过穷尽所有的可能来找到最优解

DFS

利用递归函数方便地实现暴力枚举的算法

如全排列问题

```
#include<bits/stdc++.h>
using namespace std;
const int N=105;
int n,vis[N];
vector<int> v;
void dfs(){
    if(v.size()==n){
        for(int i=0;i<v.size();i++){
            cout<<v[i]<<" ";
        }cout<<endl;
        return  ;
    }
    for(int i=1;i<=n;i++){
        if(vis[i]==1) continue;
        v.push_back(i);
        vis[i]=1;
        dfs();
        v.pop_back();
        vis[i]=0;
    }
}
int main(){
    scanf("%d",&n);
    dfs();
}
```

树上DFS

题目大意

E 和 S 在玩博弈游戏， E 先手 S 后手。

给一个 n 个节点的树，节点的值包含 $[1,n]$ 中的每一个值。

E 先随便选择一个点，占领它（点 u ）， S 只能选择与这个点相邻的，没有被占领的点（点 v ）且这两个点满足 $u \oplus v \leq min(u,v)$

$\oplus$ 是异或操作

现在把树给 E 了， E 想要重新排列节点的值（树的形状不变，只是调换结点的值）来达到这个目的：

最大化第一轮能够选择的点的数量，在选了这个点之后，E 必赢。

赢：对手无路可走，你就算赢

```
#include<bits/stdc++.h>
using namespace std;
const int N=1e6+10;
int t,n;
vector<int> v[N],bl,v1,v2,k;
int col[N],used[N],ans[N];
void dfs(int x,int pre,int fl){
    col[x]=fl;
    if(fl==0) v1.push_back(x);
    else v2.push_back(x);
    for(int i=0;i<v[x].size();i++){
        int to=v[x][i];
        if(to==pre) continue;
        int nfl=(fl==0?1:0);
        dfs(to,x,nfl);
    }
}
int main(){
    scanf("%d",&t);
    while(t--){
        scanf("%d",&n);
        v1.clear();v2.clear();k.clear();
        for(int i=1;i<=n;i++) {
            col[i]=-1;
            v[i].clear();
            used[i]=0;
            ans[i]=0;
        }
        for(int i=1;i<n;i++){
            int a,b;scanf("%d%d",&a,&b);
            v[a].push_back(b);v[b].push_back(a);
        }
        dfs(1,0,0);
        int sz=min(v1.size(),v2.size());
        for(int i=0;i<=18;i++){
            if(sz&(1<<i)){
                for(int j=(1<<i);j<(1<<(i+1));j++){
                    k.push_back(j);
```

```
                    used[j]=1;
                }
            }
        }
        if(v1.size()==sz) for(int i=0;i<sz;i++) ans[v1[i]]=k[i];
        else for(int i=0;i<sz;i++) ans[v2[i]]=k[i];
        k.clear();for(int i=1;i<=n;i++) if(!used[i]) k.push_back(i);
        int pos=0;
        for(int i=1;i<=n;i++){
            if(!ans[i]) ans[i]=k[pos++];
        }
        for(int i=1;i<=n;i++){
            cout<<ans[i]<<" ";
        }cout<<endl;
    }
}
```

BFS

又称宽度优先，就是每次都尝试访问同一层的节点。 如果同一层都访问完了，再访问下一层。

最简单的844. 走迷宫 - AcWing题库

给定一个 $n \times m$ 的二维整数数组，用来表示一个迷宫，数组中只包含 0 或 1，其中 0 表示可以走的路，1 表示不可通过的墙壁。

最初，有一个人位于左上角 $(1,1)$ 处，已知该人每次可以向上、下、左、右任意一个方向移动一个位置。

请问，该人从左上角移动至右下角 (n,m) 处，至少需要移动多少次。

数据保证 $(1,1)$ 处和 (n,m) 处的数字为 0，且一定至少存在一条通路。

输入格式

第一行包含两个整数 n 和 m。

接下来 n 行，每行包含 m 个整数（0 或 1），表示完整的二维数组迷宫。

输出格式

输出一个整数，表示从左上角移动至右下角的最少移动次数。

数据范围

$1 \le n,m \le 100$

输入样例：

```
5 5
0 1 0 0 0
0 1 0 1 0
0 0 0 0 0
0 1 1 1 0
0 0 0 1 0
```

输出样例：

```
8
```

```
#include<bits/stdc++.h>
using namespace std;
const int N=105;
int dx[]={0,0,1,-1},dy[]={1,-1,0,0};
int n,m,mp[N][N],vis[N][N];
struct node{
    int x,y,z;
};
queue<node> q;
bool check(int x,int y){
    if(x<=0||x>n||y<=0||y>m||vis[x][y]) return 0;
    if(mp[x][y]==1) return 0;
    return 1;
}
void bfs(){
    q.push({1,1,0});
    vis[1][1]=1;
    while(q.size()){
        node tp=q.front();
        if(tp.x==n&&tp.y==m){
            cout<<tp.z<<endl;
```

```
            return ;
        }
        q.pop();
        for(int i=0;i<=3;i++){
            int tx=tp.x+dx[i],ty=tp.y+dy[i];
            if(check(tx,ty)){
                q.push({tx,ty,tp.z+1});
                vis[tx][ty]=1;
            }
        }
    }

}
int main(){
    scanf("%d%d",&n,&m);
    for(int i=1;i<=n;i++) for(int j=1;j<=m;j++) scanf("%d",&mp[i][j]);
    bfs();
}
```

Dijkstra 算法

```
void dj(int st,int n,int cos){
    priority_queue<qnode> q;
    for(int i=1;i<=n;i++){
        dis[i]=INF;vis[i]=0;
    }
    while(q.size()) q.pop();
    q.push({st,0});dis[st]=0;
    while(q.size()){
        auto tp=q.top();q.pop();
        int u=tp.v;
        if(vis[u]==1) continue;
        vis[u]=1;
        if(dis[u]+cos>ans) break;
        for(int i=head[u];~i;i=d[i].nxt){
            int to=d[i].v;
            if(!vis[to]&&dis[to]>dis[u]+d[i].w){
                dis[to]=dis[u]+d[i].w;
                q.push({to,dis[to]});
            }
        }
    }
}
```

向量（vector）

叫动态数组或者变长数组应该都可以的。

特点：非常重要的数据结构，即有数组性质，也能很方便的完成离散化等操作

声明方法

①vector<数据类型>向量名(向量长度) ②vector<数据类型>向量名 ③vector<数据类型>向量名(向量长度,初始化值)

基本操作

```
push_back(x)//将x添加至末尾

pop_back()//删除最后一个数

front() //返回最后一个元素

back()//返回最后一个元素

size()//返回向量中元素的个数

insert(x,y)//在x位置插入元素（注意x通常由迭代器指定）
```

对（pair）

可以将两个数据组合成一组，实现方式是结构体。

特点：最简单的容器，可以很简单的组合两个元素，在数据类型组合非常多的情况下可以代替struct压行。

```
pair<T1,T2>p1; //声明一个pair对象p1,first类型为T1, second类型为T2

pair<int,double>p2=make_pair(v1,v2) //创建v1和v2组合成的pair对象

pair<string,int>name_age("Tom",18);//第三种创建方法

cout<<name_age.first<<""<<name_age.second<<endl;//
```

链表（list）

万年没用过了，大家可以跳过这个

特点：不能随意访问任意元素，只能从表头循环遍历

声明方式

①list<数据类型>数据名; ②list<数据类型>数据名(初始长度); ③list<数据类型>数据名(初始长度,初始值);

基本操作

```
push_back(x)//表尾添加元素x

pop_back()//删除表尾元素

push_front(x)//表头添加元素x

pop_front()//删除表头元素

erase(x)//删除迭代器x位置的元素

remove(x)//删除值为x的元素
```

集合（set）

特点：无重复元素，增删元素后会自动排序

声明方式：set<数据类型>集合名;

基本操作

```
clear()//清除所有元素

count(x)//返回集合中x的个数（是否存在）

size()//返回集合中元素个数

empty()//判断set容器是否为空

insert(x)//将x插入集合中

erase(x)//将x从集合中删除

find(x)//返回指向x的迭代器
```

关于效率

map和set的插入删除效率比其他序列容器高。set的元素是以节点方式存储的，节点的结构和链表差不多，指向父节点和子节点。set的查找使用的是二分，在logn复杂度内能够找到。

相关操作

```
prev  lower_bound upper_bound
```

多重集合(multiset)

近视理解为元素可重复的set，声明方式:multiset<数据类型>集合名

映射（map）

声明方式：map<数据类型1，数据类型2>映射名

特点：红黑树实现，内部元素按字典序，每种元素只能存在一个。对空间的占用较高，但是对查找、插入操作都很高效。

基本操作：

```
map<string,int>mp;

mp["字符串"]=2412;

is=mp.find(x);//返回键为x的地址，若没有，则返回end()即最后一个映射关系的地址

for(map<string,int>::iteratorit=mp.begin();it!=mp.end();it++){

       cout<<(*it).first<<""<<(*it).second<<endl;

}
```

unordered_map

效率更高的一种map，记录元素的hash值，根据hash判断元素是否相同。

特点：内部元素无序

栈（stack）

声明方法：stack<数据类型>栈名 特点：先进后出 基本操作：

```
push(x)—将x进栈
pop()—删除栈顶元素
top()—返回栈顶元素
size()—返回栈中元素个数
empty()—返回栈是否为空
```

队列（queue）

声明方法：queue<数据类型>队列名; 特点：先进先出

基本操作

```
push(x)//将x入队

pop()//弹出队列第一个元素

front()//返回队头元素

back()//返回队尾元素

empty()//返回队伍是否为空

size()//返回队伍元素个数
```

优先队列（priority_queue）

增删元素后自动排序的队列，默认从大到小

声明方式

①从小到大的队列：priority_queue< int,vector ,greater > q1; ①从大到小的队列：priority_queue< int,vector ,less > q2;

基本操作

```
push(x)//将x入队

pop()//删除队首元素

top()//返回队顶元素

size()//返回优先队列中元素个数

empty()//返回队列是否为空
```

重定义排序相关操作

```
bool operator<(const node &b) const{

}
```

双向队列（deque）

特点

双端插入和删除

#####基本操作

```
deque.size()

deque.empty()

deque.push_back()//向后插入

deque.push_front()//向前插入

deque.pop_back() //后部弹出

deque.pop_front()//前部弹出

deque.front()

deque.back()

deque.resize(num,elem); //重新指定容器的长度为num,若容器变长，则以elem值填充新位置,如果容器变短，则末尾超出容器长度的元素被删除。

迭代器

for(deque<int>::const_iteratorit=d.begin();it!=d.end();it++){

        cout<<*it<<" ";

}
```