广度优先搜索 2

在上一讲中，重点介绍了广搜的模板，及其在棋盘上的应用，主要包括求连通块和无权最短路径。在这一讲中，主要介绍广搜求最短路在其他一些题型上的应用。如果在题目中看到类似最少步骤、最短时间等元素，可以优先从广搜的角度思考解题策略。通过以下题目，也应该熟练掌握广搜的模板：

```
初始状态入队
while(队列不为空){
        取出队头结点;
        for(枚举队头结点的拓展状态){
                if(该拓展状态可用){
                        if(该拓展状态是目标状态){
                                输出答案;
                                return;
                        }
                        锁定该拓展转态;
                        拓展转态入队;
                }
        }
}
```

砝码称重

【题目描述】
给定 n 种砝码(每种砝码个数不限)和一个整数 M,求至少需要几个砝码才可以称出刚好 M 克。
【输入】
第一行：N,M。(n<=100,m<=1000)
第二行：每种砝码的重量 w[i]。
【输出】
一行:称出重量 m 最少需要的砝码个数；如果称不出重量 m 则输出“Impossible”
【样例输入】
3 4
1 2 3
【样例输出】
2
【分析】
本题的解法较多，可以用完全背包的方法来解，也可以用广搜来解。我们假设初始状态为 0，步骤也为 0，则选择一个砝码（以样例为例），可以达到 1、2、3 的状态。下一次，以 1 为队头结点，再加上一个砝码，可以拓展到 2、3、4 的状态，但是这里 2、3 这个状态已经拓展过了，不入队，4 即为所求状态。每次从队头结点一层一层拓展，最先达到所求目标，则其步骤也是最短的，即所需砝码最少。
【代码】

```cpp
#include<iostream>
#include<queue>
using namespace std;
struct node{
        int x,s;    //x 表示达到的状态，s 表示所需的砝码数
};
queue<node> q;
int n,m,a[101];
bool vis[1001];    //标记一个状态是否访问过
int main(){
        cin>>n>>m;
        for(int i=1;i<=n;i++)cin>>a[i];
        q.push((node){0,0});    //初始状态入队
        vis[0]=true;    //标记该状态访问过
        while(!q.empty()){
                node f=q.front();    //取得队头结点
                q.pop();
                for(int i=1;i<=n;i++){  //由队头结点枚举拓展状态
                        int v=f.x+a[i];  //加上第 i 个砝码能达到的状态
                        if(vis[v]==false&&v<=m){//如果该状态没出现过且小于等于 m
                                if(v==m){    //如果达到所求目标
                                        cout<<f.s+1;  //输出步骤
                                        return 0;    //终止程序
                                }
                                q.push((node){v,f.s+1});    //拓展状态入队
                                vis[v]=true;    //标记该状态入队过
                        }
                }
        }
        cout<<"Impossible"<<endl;    //如果没有中途结束，说明找不到方案
}
```

抓牛

【题目描述】
农夫约翰被通知，他的一只奶牛逃逸了！所以他决定，马上出发，尽快把那只奶牛抓回来.他们都站在数轴上．约翰在 N(0≤N≤100000)处，奶牛在 K(0≤K≤100000)处．约翰有两种办法移动，步行和瞬移：步行每秒种可以让约翰从 x 处走到 x+1 或 x-1 处；而瞬移则可让他在 1 秒内从 x 处消失，在 2x 处出现．然而那只逃逸的奶牛，悲剧地没有发现自己的处境多么糟糕，正站在那儿一动不动．
那么，约翰需要多少时间抓住那只牛呢？
【输入】
仅有两个整数 N 和 K
【输出】

最短时间

【样例输入】

5 17

【样例输出】

4

【分析】

本题是给出一个起点和终点，求从起点到终点的最短时间，很明显的广搜题。将起点入队，按照时间顺序一层一层往外拓展，难点是一个状态怎样拓展成其他状态。详见代码。

【代码】

```
#include<iostream>
#include<queue>
using  namespace  std;
struct  point{
        int  p,t;     //p 表示位置，t 表示到达 p 位置所需的时间
};
queue<point>  q;
int  n,k;
bool  vis[200000];     //标记位置有没有走过
int  main(){
        cin>>n>>k;
        if(n==k){
                cout<<0;
                return  0;
        }
        q.push((point){n,0});     //初始状态入队
        vis[n]=true;     //标记该点走过
        while(!q.empty()){
                point  f=q.front();     //取得队头结点
                q.pop();
                if(f.p+1==k||f.p-1==k||f.p*2==k){     //拓展一步到终点
                        cout<<f.t+1;     //输出时间
                        return  0;     //结束程序
                }
                if(f.p<k&&!vis[f.p+1]){     //判断能否走到 x+1 处
                        q.push((point){f.p+1,f.t+1});  //可以走，入队
                        vis[f.p+1]=true;
                }
                if(f.p>0&&!vis[f.p-1]){     //判断能否走到 x-1 处
                        q.push((point){f.p-1,f.t+1});     //可以走，入队
                        vis[f.p-1]=true;
                }
                if(f.p<k&&!vis[f.p*2]){     //判断能否走到 2*x 处
                        q.push((point){f.p*2,f.t+1});     //可以走，入队
```

```
35.                         vis[f.p*2]=true;
36.                 }
37.         }
38. }
```

奇怪的电梯

【题目描述】
呵呵，有一天我做了一个梦，梦见了一种很奇怪的电梯。大楼的每一层楼都可以停电梯，而且第 i 层楼(1<=i<=N)上有一个数字 Ki(0<=Ki<=N)。电梯只有四个按钮：开，关，上，下。上下的层数等于当前楼层上的那个数字。当然，如果不能满足要求，相应的按钮就会失灵。例如：3 3 1 2 5 代表了 Ki(K1=3,K2=3,……)，从一楼开始。在一楼，按“上”可以到 4 楼，按“下”是不起作用的，因为没有-2 楼。那么，从 A 楼到 B 楼至少要按几次按钮呢？
【输入】
输入文件共有二行，第一行为三个用空格隔开的正整数，表示 N,A,B(1≤N≤200, 1≤A,B≤N)，第二行为 N 个用空格隔开的正整数，表示 Ki。
【输出】
输出文件仅一行，即最少按键次数，若无法到达，则输出-1。
【样例输入】
5 1 5
3 3 1 2 5
【样例输出】
3
【分析】
本题与上题类似，也是给出一个起点，一个终点，求从起点到终点所需的最少步数，广搜处理。
【代码】

```
1.  #include<iostream>
2.  #include<queue>
3.  using  namespace  std;
4.  struct  point{
5.          int  p,s;     //p 表示当前楼层，s 表示到达当前楼层的按键数
6.  };
7.  queue<point>  q;
8.  int  n,a,b,floor[201];
9.  bool  vis[201];
10. int  main(){
11.         cin>>n>>a>>b;
12.         for(int  i=1;i<=n;i++)cin>>floor[i];
13.         if(a==b){     //如果可以直接到，输出 0
14.                 cout<<0;
15.                 return  0;
16.         }
```

```
17.         q.push((point){a,0});     //初始状态入队
18.         vis[a]=true;
19.         while(!q.empty()){
20.                 point  f=q.front();     //取出队头结点
21.                 q.pop();
22.                 int  up=f.p+floor[f.p];     //往上到达的楼层
23.                 int  down=f.p-floor[f.p];     //往下到达的楼层
24.                 if(up==b||down==b){     //如果能到达目标楼层
25.                         cout<<f.s+1;     //输出按键数
26.                         return  0;     //结束程序
27.                 }
28.                 if(up<=n&&vis[up]==false){     //如果可以往上
29.                         q.push((point){up,f.s+1});     //入队
30.                         vis[up]=true;
31.                 }
32.                 if(down>=1&&vis[down]==false){     //如果可以往下
33.                         q.push((point){down,f.s+1});     //入队
34.                         vis[down]=true;
35.                 }
36.         }
37.         cout<<-1;     //到不了，输出-1
38. }
```

程序龙的游戏

【题目描述】

在 oiclass，程序龙喜欢和小朋友们一起玩游戏，今天，他有想到了一个好玩的游戏。

游戏规则：每次点击一个小朋友，他和他的周围的小朋友都会改变状态（蹲下的变成了站起来的，站起来的变成了蹲下的）

我们将这个抽象成如下图所示的 1*N 的图。对于一个单元格,黑色表示小朋友是站起来的，反之，蹲下的小朋友是白色。Source 表示初始状态,Target 表示目标状态。



Source



Target

现在程序龙有点偷懒,希望玩游戏的你算出初始状态到目标状态的最少点击数。

【输入】

第一行为 N 表示小朋友的个数

第二行是初始状态,有 N 个数，每个数不是 0 就是 1.(0 表示小朋友是蹲下的,1 表示小朋友是站起来的)

第三行的结构跟第二行类似，表示目标状态

【输出】
一个数 X,表示初始状态到目标状态的最少点击数。
如果无法到达目标,则请输出"Boring"
【样例输入】
9
0 1 0 0 0 1 0 0 0
1 0 1 0 1 0 1 0 0
【样例输出】
2
【提示】
对于 100%的数据，N<=10
【分析】
本题也是给出一个起始状态和一个目标状态，求从起始状态到目标状态的最少步骤，广搜解决。难点在于，如何将一个状态拓展成其他状态。根据题意，我们可以将一个状态用二进制数表示，对于二进数中的 n 个位，一旦点击一个位，则它和相邻位进行取反操作，这个操作我们可以通过异或 111（二进制）来实现，比如 01010101 这个数，我们现在点击右数第 4 位 0，则其状态改变为 01001001，这个操作等价于（01010101）^(00011100),而（00011100）其实就是 7<<2，那么，点击第 i 位，就是将状态 x^(7<<(i-2))，枚举每一位，就可以得到状态 x 的所有拓展状态。当然，对于首位和末位要特殊处理。点击首位可以直接异或 3,末位可以异或（3^(n-2)）。
【代码】

```
#include<iostream>
#include<queue>
using  namespace  std;
struct  point{
        int  x,s;     //x 表示一个二进制状态，s 表示到达这个状态所需的步数
};
queue<point>  q;
int  n,a,s,t;
bool  vis[1025];
void  check(int  x,int  s){     //判断状态 x 能否入队
        if(vis[x]==false){     //如果该状态没访问过
                if(x==t){     //该状态是目标状态
                        cout<<s+1;     //输出步骤
                        exit(0);     //终止程序
                }
                q.push((point){x,s+1});     //该状态入队
                vis[x]=true;     //标记该状态访问过
        }
}
int  main(){
        cin>>n;
        for(int  i=0;i<n;i++){
```

```
23.                 cin>>a;
24.                 s=(s<<1)+a;     //将起始状态表示成一个二进制数
25.         }
26.         for(int  i=0;i<n;i++){
27.                 cin>>a;
28.                 t=(t<<1)+a;     //将目标状态表示成一个二进制数
29.         }
30.         if(s==t){     //起始状态和目标状态相同
31.                 cout<<0;     //直接输出 0
32.                 return  0;
33.         }
34.         q.push((point){s,0});     //初始状态入队
35.         vis[s]=true;     //标记该状态访问过
36.         while(!q.empty()){
37.                 point  f=q.front();  //取得队头结点
38.                 q.pop();
39.                 int  nx=f.x^3;     //点击首位
40.                 check(nx,f.s);     //判断该状态可能入队
41.                 for(int  i=0;i<n-2;i++){     //枚举点击第 2 到第 n-1 位
42.                         int  t=(7<<i);
43.                         nx=f.x^t;     //得到点击后的状态
44.                         check(nx,f.s);     //判断该状态能否入队
45.                 }
46.                 nx=f.x^(3<<(n-2));     //点击第 n 位
47.                 check(nx,f.s);     //判断该状态能否入队
48.         }
49.         cout<<"Boring";     //达不到目标状态
50. }
```

素数路径

【题目描述】

内阁的部长们对于安全部门首脑声称要改变他们办公室的 4 个房间号非常烦恼。

—这只是一件安全措置，不时地改变这样的事物，使得敌人处于盲区。

— 但是你看，我已经选了我的房间号 1033 有很好的理由，我是总理，你知道的。

— 我知道的，所以你的新房间号 8179 也是素数，你只要在你的办公室房门上将这新的 4 位数字贴在老的 4 位数字上就可以了。

— 不，不那么简单。假如我把第一个数字改成 8，那么数字 8033 就不是素数！

— 我知道，作为总理，你不能忍受一个非素数作为你的房间号即使只有几秒钟。

— 正确！我必须找到一个从 1033 到 8179 的一个素数路径的方案，从一个素数到下一个素数只要改变一位数。

一直在听的财政部长，也介入了讨论。

— 请不要产生不必要的开支！我碰巧知道，改一位数字的价格是 1 英镑。

— 在这种情况下，我需要一个计算机程序使得花费最小化，你知道有非常便宜的软件开发

者吗？

— 我知道。有个程序设计竞赛马上要进行了。让他们帮助总理在两个四位素数之间找到最便宜的素数路径。当然，第一位是非零的。上述实例有个解答：

1033
1733
3733
3739
3779
8779
8179

这个解答花费 6 英镑。注意第 1 个位置在第 2 步被粘贴，在最后一步不能被重用，在第 1 个位置上要粘贴新值必须购买。

【输入】

第一行给出一个正整数：测试用例的数目（最多 100）。每个测试用例一行，两个用空格分开的数字，这两个数字都是 4 位素数（不以 0 作为首位）。

【输出】

对于每个测试用例，输出一行，或者是最小花费的数目，或者输出 Impossible。

【样例输入】

3
1033 8179
1373 8017
1033 1033

【样例输出】

6
7
0

【分析】

本题是给出一个起始素数，根据指定规则转换成一个目标素数，所需的最少步骤，广搜。

【代码】

```
#include<iostream>
#include<queue>
#include<cstring>
#include<cmath>
using namespace std;
struct point{
    int x,s;
};
bool prime[10001],used[10001];
int s,t,n;
void init(){  //统计出 10000 以内的素数表
    memset(prime,true,sizeof(prime));
    prime[0]=prime[1]=false;
    for(int i=2;i<=10000;i++){  //用筛法求素数
```

oiclass

```
        if(prime[i]==true){  //如果 i 是素数
            for(int j=2;j<=10000/i;j++){  //则将 i 的倍数
                prime[i*j]=false;  //标记为不是素数
            }
        }
    }
}
int bfs(int s,int t){
    if(s==t)return 0;  //如果是目标状态，直接输出 0
    memset(used,false,sizeof(used));  //因为是对样例输入，记得初始化数组
    queue<point> q;  //在函数里声明队列，省略了队列的初始化，但是有栈溢出风险
    q.push((point){s,0});  //初始状态入队
    used[s]=true;  //标记该状态访问过
    while(!q.empty()){
        point f=q.front();  //取出队头结点
        q.pop();
        for(int i=0;i<4;i++){  //外循环枚举每一位数，总 4 位数
            int pwr=pow(10,i);
            int mid=f.x/pwr%10;  //获得第 i+1 位数
            for(int j=0;j<=9;j++){  //枚举该位数可换成 0 到 9 的任意一位
                int nx=f.x-mid*pwr+j*pwr;  //合成新的状态
                if(prime[nx]&&!used[nx]&&nx>999){//是素数且没用过且首位不为 0
                    if(nx==t){  //该状态是目标状态
                        return f.s+1;  //返回步骤
                    }
                    used[nx]=true;  //标记该数访问过
                    q.push((point){nx,f.s+1});  //入队
                }
            }
        }
    }
    return -1;  //如果没有中途返回，说明无法转化成目标状态，返回-1
}
int main(){
    init();
    cin>>n;
    for(int i=1;i<=n;i++){
        cin>>s>>t;  //输入起始状态和目标状态
        int ans=bfs(s,t);  //广搜求出最少步骤
        if(ans==-1)cout<<"Imposible"<<endl;
        else cout<<ans<<endl;
    }
}
```

最小数字倍数

【题目描述】
写一个程序，对于给定的一个自然数N(0≤N≤4999)，以及M个不同的十进制数字X1,X2...XM（M≥1 最多 10 个数字），找出一个由 X1,X2..XM 组成的整数 P，使得 P 是 N 的严格的倍数，且要求 P 最小。
【输入】
第一行为一个整数 N。第二行为一个整数 M。此后的 M 行分别为 M 个数字 X1,X2..XM。
【输出】
输出一个整数 P（占一行），保证有解。
【样例输入】
22
3
7
0
1
【样例输出】
110
【分析】
本题是一道藏得比较深的广搜题，没有起始状态，没有目标状态，要求 n 的最小倍数，我们可以按照数字从小到大排列，每一层为一位数，每一层往外扩展，最先找到的能够整除 n 的数，即 n 的最小倍数，这其实就是广搜的思想。大概大码如下：

```
#include<iostream>
#include<queue>
#include<algorithm>
using namespace std;
queue<long long> q;
int n,m,a[11];
int main(){
    cin>>n>>m;
    for(int i=1;i<=m;i++)cin>>a[i];
    sort(a+1,a+m+1);  //将数据从小到大排序
    q.push(0);  //初始状态入队
    while(!q.empty()){
        long long f=q.front();  //取出队头结点
        q.pop();
        for(int i=1;i<=m;i++){
            long long np=f*10+a[i];  //最队头的基础上组接多一位数
            if(np%n==0&&f!=0){  //如果该数能够被 n 整除且不为 0
                cout<<np<<endl;  //输出结果
                return 0;
            }
            q.push(np);  //新数入队，准备出队时再组接一位数
```

```
        }
    }
}
```

以上代码朴素可以理解，但是会超时。为什么呢？很简单，每层都拓展 n 次，对于第 n 层，拓展的结点数达到 n 的 n 次方。我们可以进行优化。如果三位数 x%n==2，和一个四位数 y%n==2，这两个数都需要加上 2 才能被 n 整除，那么最小的 n 的倍数肯定是加给三位数 x，所以，如果一个小数%n 的余数出现了，后续在出现%n 的余数，则不需入队处理。优化后的代码如下：

```
#include<iostream>
#include<queue>
#include<algorithm>
using namespace std;
struct point {
    long long p;  //表示组接的新数
    int r;  //表示新数对 n 的余数
};
queue<point> q;
int n,m,a[11];
bool used[5000];
int main() {
    cin>>n>>m;
    for(int i=1; i<=m; i++)cin>>a[i];
    sort(a+1,a+m+1);
    q.push((point) {0,0});
    while(!q.empty()) {
        point f=q.front();
        q.pop();
        for(int i=1; i<=m; i++) {
            long long np=f.p*10+a[i];
            if(!used[np%n]) {  //判断该余数没有出现过
                if(np%n==0&&f.p==0)continue;  //判断前导零问题
                if(np%n==0) {
                    cout<<np<<endl;
                    return 0;
                }
                q.push((point){np,np%n});
                used[np%n]=true;  //标记该余数出现过
            }
        }
    }
}
```

然而以上代码还是不能通过本题，为什么呢？因为组成 n 的倍数，可能需要一个很长的数，

这个数超过 long long 的存储范围。当然，可能可以想到用字符串来存储这个数，但是这样空间消耗会较大，但也不为一个解决问题的办法。

```cpp
#include<iostream>
#include<queue>
#include<algorithm>
#include<string>
#include<sstream>
using namespace std;
struct point {
    string p;  //存储合成的数，字符创类型
    int r;
};
queue<point> q;
int n,m,a[11];
bool used[5000];

int main() {
    cin>>n>>m;
    for(int i=1; i<=m; i++)cin>>a[i];
    sort(a+1,a+m+1);
    q.push((point) {"",0});
    while(!q.empty()) {
        point f=q.front();
        q.pop();
        for(int i=1; i<=m; i++) {
            long long nr=(f.r*10+a[i])%n; //只需将上次的余数加上新数再求余
            if(!used[nr]) {  //如果该余数没用过
                if(nr%n==0&&f.p==""){  //判断前导零的情况
                    continue;
                }
                string s;  //以下代码将 a[i]转换成字符创
                stringstream ss;
                ss<<a[i];
                ss>>s;
                string np=f.p+s;  //将原数合并上 a[i]形成新数
                if(nr%n==0) {  //如果可以被整除
                    cout<<np<<endl;
                    return 0;
                }
                q.push((point){np,nr});  //将新数入队
                used[nr]=true;  //标记余数用过
            }
        }
    }
```

```
}
```

其实正解的方法是队列中不存储整个数，只需要存储当前位数就可以，同时，队列中的每一个元素都有一个指针，标记该元素是由队列中的那个位置拓展来的，最后输出合成的数时，沿着前置指针往前回溯一一输出即为合成的数，当然，如果这样，我们就无法用内置的 queue 模板，需要自己手动写一个队列，好在队列可以直接用数组模拟实现，增加两个指针 f 和 r 指向队头和队尾即可。代码如下，仔细体会。

【代码】

```
#include<iostream>
#include<algorithm>
using namespace std;
struct point{
    int di,r,pre;  //di 表示一个数，r 表示余数，pre 表示由那个队列元素拓展来
}q[10000];  //自定义的队列
int n,m,a[11],f,r;
bool used[5000];
void print(int ind){  //打印出合成的新数，递归实现
    if(q[ind].pre==-1)return;  //遇到-1 终止
    print(q[ind].pre);  //先输出该数的前一个数
    cout<<q[ind].di;  //输出该数
}
int main(){
    cin>>n>>m;
    for(int i=1;i<=m;i++)cin>>a[i];
    sort(a+1,a+1+m);
    q[++r]=(point){0,0,-1};  //初始状态入队，-1 表示起始状态
    while(f<r){
        point now=q[++f];  //取出队头结点
        for(int i=1;i<=m;i++){  //由队头结点进行拓展
            int nx=(now.r*10+a[i])%n;  //计算新的余数
            if(!used[nx]){  //该余数没用过
                if(nx==0&&q[f].r==0)continue;  //判断前导零
                used[nx]=true;  //标记该余数用过
                q[++r]=(point){a[i],nx,f};//入队，当前元素 r 由 f 拓展来的
                if(nx==0){  //如果能够被整除
                    print(r);  //输出合成的数
                    return 0;
                }
            }
        }
    }
}
```