BFS

Flood Fill

`可以在线性时间内，找到连通块`

池塘计数

AC 代码

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

const int N = 1010;

int n,m,ans;
int dx[] = {1,0,-1,0,1,1,-1,-1};
int dy[] = {0,1,0,-1,1,-1,-1,1};
char mp[N][N];
bool st[N][N];

typedef pair<int,int> PII;

void bfs(int x,int y)
{
    queue<PII> q;
    q.push({x,y});
    mp[x][y] = 0;
    st[x][y] = 1;
    int xx,yy;
    while(q.size())
    {
        auto t = q.front();
        q.pop();

        for(int i=0;i<8;i++)
        {
            xx = t.first + dx[i];
            yy = t.second + dy[i];

            if(xx>=1 && xx<=n && yy>=1 && yy<=m
            && mp[xx][yy]==1 && !st[xx][yy])
            { //检查是否为连通块
                mp[xx][yy] = 0;
                st[xx][yy] = true;
                q.push({xx,yy});
            }
        }

    }
}
```

```
int main()
{
    cin>>n>>m;
    for(int i=1;i<=n;i++)
        for(int j=1;j<=m;j++)
        {
            char t;
            cin>>t;
            if(t=='W')
                mp[i][j] = 1;
        }

    for(int i=1;i<=n;i++)
    {
        for(int j=1;j<=m;j++)
        {
            if(mp[i][j]==1)
            {
                bfs(i,j);
                ans++;
            }
        }
    }

    cout<<ans;
}
```

[城堡问题]

AC代码

```
/*
每个方块中墙的特征由数字 P 来描述，我们用1表示西墙，2表示北墙，4表示东墙，8表示南墙，P 为该方块包含墙的数字之和。
例如，如果一个方块的 P 为3，则 3 = 1 + 2，该方块包含西墙和北墙。
可以用二进制位来判断有没有墙，由于加起来和不超过 15 ,所以有四位 二进制数 ，第一位为 1 表示有西墙...
以此类推， 我们将遍历方向 与 数字 P 右移 >> & 1 所到位置墙的方向表示一致
*/
#include<bits/stdc++.h>
using namespace std;

const int N = 55;

int m,n;
int g[N][N];
bool st[N][N];

int bfs(int sx,int sy)
{
    int dx[] = {0,-1,0,1}; //与二进制位一样的方向
    int dy[] = {-1,0,1,0};
    int area = 0;

    queue<pair<int,int>>q;
    q.push({sx,sy});
    st[sx][sy] = true;
```

```
    while(q.size())
    {
        auto t = q.front();
        q.pop();
        area ++; //记录房间面积

        for(int i = 0; i < 4; i++)
        {
            int xx = t.first + dx[i] , yy = t.second + dy[i];
            if( st[xx][yy] || xx < 1 || xx > m || yy < 1 || yy > n )
continue;
            if( g[t.first][t.second] >> i & 1 ) continue; //当前位置有墙    二进制位

            q.push({xx,yy});
            st[xx][yy] = true;
        }

    }

    return area;
}

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

    for(int i = 1; i <= m; i++)
    {
        for(int j = 1; j <= n; j++)
            cin >> g[i][j];
    }

    int cnt = 0,area = 0;
    for(int i = 1; i <= m; i++)
    {
        for(int j = 1; j <= n; j++)
            if(!st[i][j])
            {
                area = max(area,bfs(i,j));
                cnt++;
            }
    }

    cout << cnt << '\n' << area << endl;

    return 0;
}
```

[山峰和山谷]

AC代码

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

const int N = 1010;
```

```cpp
int n;
int g[N][N];
bool st[N][N];

void bfs(int sx,int sy,bool & has_higher,bool & has_lower)
{
    int dx[] = {1,0,-1,0,1,1,-1,-1};
    int dy[] = {0,1,0,-1,1,-1,-1,1};

    queue<pair<int,int>>q;
    q.push({sx,sy});
    st[sx][sy] = true;

    while(q.size())
    {
        auto t = q.front();
        q.pop();

        for(int i = 0; i < 8; i++)
        {
            int xx = t.first + dx[i] , yy = t.second + dy[i];
            if( xx < 1 || xx > n || yy < 1 || yy > n )  continue;

            if(g[xx][yy] != g[t.first][t.second])
            {
                if(g[xx][yy] > g[t.first][t.second])    has_higher = true;
                else    has_lower = true;
            }
            else if(!st[xx][yy])
            {
                q.push({xx,yy});
                st[xx][yy] = true;
            }
        }
    }
}

int main()
{
    cin >> n;

    for(int i = 1; i <= n; i++)
    {
        for(int j = 1; j <= n; j++)
            cin >> g[i][j];
    }

    int valley = 0 , peak = 0;
    for(int i = 1; i <= n; i++)
    {
        for(int j = 1; j <= n; j++)
        {
            if(!st[i][j])
            {
                bool has_higher = false,has_lower = false;
                bfs(i,j,has_higher,has_lower);
                if(!has_lower)  valley++;
```

```
                if(!has_higher) peak++;
            }
        }
    }

    cout << peak << ' ' << valley << endl;

    return 0;
}
```

最短路模型

[抓住那头牛]

AC代码

```
#include<iostream>
#include<queue>
using namespace std;
//注意数组范围应该开到两倍大，因为他可以一次跳 2*t 步
const int N = 2e5 + 10;
int g[N],n,k;
bool st[N];
//注意数组不要越界，否则段错误
int bfs()
{
    queue<int>q;
    q.push(n);
    st[n] = true;

    while(q.size())
    {
        int t = q.front();
        q.pop();

        if(t == k)  return g[t];
        //枚举三种跳跃方式
        if(t + 1 < N && !st[t + 1])
        {
            g[t + 1] = g[t] + 1;
            st[t + 1] = true;
            q.push(t + 1);
        }
        if(t - 1 >= 0 && !st[t - 1])
        {
            g[t - 1] = g[t] + 1;
            st[t - 1] = true;
            q.push(t - 1);
        }
        if(t * 2 < N && !st[2 * t])
        {
            g[t * 2] = g[t] + 1;
            st[t * 2] = true;
            q.push(t * 2);
        }
```

```
    }
    return -1;
}

int main()
{
    cin >> n >> k;
    g[n] = 0;

    cout << bfs() << endl;

    return 0;
}
```

迷宫问题

AC代码

```
#include<bits/stdc++.h>
using namespace std;
//没啥好说的，直接bfs一遍，记录一下路径就可以了
#define x first
#define y second

const int N = 1010;
int mp[N][N],n;
pair<int,int>pre[N][N];

int dx[] = {1,0,-1,0};
int dy[] = {0,1,0,-1};

void bfs(int sx,int sy)
{
    queue<pair<int,int> > q;
    q.push({sx,sy});
    pre[sx][sy] = {-2,-2};
    mp[sx][sy] = 1;

    while(q.size())
    {
        auto t = q.front();
        q.pop();

        for(int i = 0; i < 4; i++)
        {
            int xx = t.x + dx[i] , yy = t.y + dy[i];
            if( xx >= 0 && xx <= n-1 && yy >= 0 && yy <= n-1
            && mp[xx][yy] == 0 )
            {
                pre[xx][yy] = {t.x,t.y};
                if(xx == n-1 && yy == n-1)  return;
                q.push({xx,yy});
                mp[xx][yy] = 1;
            }
        }
    }
```

```
}

void printpath(int u,int v)
{
    if( pre[u][v].x != -2 && pre[u][v].y != -2 )    printpath(pre[u][v].x,pre[u][v].y);
    cout << u << ' ' << v << endl;
}

int main()
{
    cin >> n;

    memset(pre,-1,sizeof pre);

    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            cin >> mp[i][j];

    bfs(0,0);

    printpath(n-1,n-1);

    return 0;
}
```

多源BFS

多个源点开始搜

[矩阵距离]

AC代码

```
#include<bits/stdc++.h>
using namespace std;
//BFS的两段性和单调性保证了，每次更新的点都是最短的
const int N = 1010;
char g[N][N];
int n,m,dist[N][N];

typedef pair<int,int> PII;

void bfs()
{
    int dx[] = {0,1,-1,0};
    int dy[] = {1,0,0,-1};

    queue<PII>q;

    memset(dist,-1,sizeof dist);

    for(int i = 0; i < n; i++) //多起点源bfs
        for(int j = 0; j < m; j++)
            if(g[i][j] == '1')  q.push({i,j}),dist[i][j] = 0;
```

```
    while(q.size())
    {
        PII t = q.front();
        q.pop();

        for(int i = 0; i < 4; i++)
        {
            int xx = t.first + dx[i] , yy = t.second + dy[i];

            if( xx < 0 || xx >= n || yy < 0 || yy >= m )    continue;
            if( dist[xx][yy] != -1 )    continue; //更新过，已经是最短
            //BFS队列里有 两段性 和 单调性 故每次更新的都是最短距离
            dist[xx][yy] = dist[t.first][t.second] + 1; // 每次只往外扩展一层
            q.push({xx,yy});
        }
    }
}

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

    for(int i = 0; i < n; i++)  cin >> g[i];

    bfs();

    for(int i = 0; i < n; i++)
    {
        for(int j = 0; j < m; j++)
            cout << dist[i][j] << ' ';
        cout << endl;
    }

    return 0;
}
```

最小步数模型

[魔板]

AC代码

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

char g[2][4];
unordered_map<string,int>dist; //存最小花费（步数）
unordered_map<string,pair<char,string> >pre; //存路径与选择

void set(string str) //转化为矩阵字符
{
    int k = 0;
```

```
    for(int i = 0; i < 4; i++)  g[0][i] = str[k++];
    for(int i = 3; i >=0; i--)  g[1][i] = str[k++];
}

string get() //转化为行字符
{
    string res;
    for(int i = 0; i < 4; i++)  res += g[0][i];
    for(int i = 3; i >= 0; i--) res += g[1][i];

    return res;
}

string opt_A(string str) //上下交换
{
    set(str);

    for(int i = 0; i < 4; i++)  swap(g[0][i],g[1][i]);

    return get();
}

string opt_B(string str) //右移，最后一列变第一列
{
    set(str);
    char e1 = g[0][3] , e2 = g[1][3];

    for(int i = 2; i >= 0; i--)
        g[0][i+1] = g[0][i],g[1][i+1] = g[1][i];

    g[0][0] = e1,g[1][0] = e2;
    return get();
}

string opt_C(string str) //旋转
{
    set(str);

    char t = g[0][1];
    g[0][1] = g[1][1];
    g[1][1] = g[1][2];
    g[1][2] = g[0][2];
    g[0][2] = t;

    return get();
}

void bfs(string start,string end)
{
    queue<string> q;
    q.push(start);

    dist[start] = 0;
    if(start == end)    return; // 初始既是答案。。

    while(q.size())
    {
        string t = q.front();
```

```
        q.pop();

        string m[3]; // 三种操作

        m[0] = opt_A(t);
        m[1] = opt_B(t);
        m[2] = opt_C(t);

        for(int i = 0; i < 3; i++)
        {
            string u = m[i];
            if(dist.count(u))   continue; //该种状态已更新最小花费

            dist[u] = dist[t] + 1; // 更新最小花费
            pre[u] = {char(i + 'A'),t};
            if(u == end)    return; // 到达答案
            q.push(u);
        }
    }

}

void print_opt(string &end,string &start)
{
    if(end == start)    return;// 初始就是结束，无选择

    if(pre[end].second != "12345678")   print_opt(pre[end].second,start);
    cout << pre[end].first; //递归求路径
}

int main()
{
    string start = "12345678",end;

    for(int i = 0; i < 8; i++)
    {
        int x;
        cin >> x;
        end += char(x + '0');
    }

    bfs(start,end);

    cout << dist[end] << endl;

    print_opt(end,start);

    return 0;
}
```

双端队列BFS

[电路维修]

当路径的权值有两种情况时，使用双端队列BFS，保持BFS的两段性和单调性，将权值小的加到队头，权值大的加到队尾。（和Dijkstra堆优化版有点像）

AC代码

```
#include<iostream>
#include<cstring>
#include<deque>
using namespace std;

const int N = 510;
typedef pair<int,int> PII;
#define x first
#define y second

char g[N][N];
int n,m,dist[N][N];
bool st[N][N];

int bfs()
{
    int dx[] = {-1,-1,1,1} , dy[] = {-1,1,1,-1};
    int ix[] = {-1,-1,0,0} , iy[] = {-1,0,0,-1};
    char dir[5] = "\\/\\/";      //按顺序枚举每种情况

    deque<PII> q;
    memset(dist,0x3f,sizeof dist);
    memset(st,0,sizeof st);
    dist[0][0] = 0;
    q.push_front({0,0});

    while(q.size())
    {
        PII t = q.front();
        q.pop_front();

        if(st[t.x][t.y])    continue;
        st[t.x][t.y] = true;
        if(t.x == n && t.y == m)    return dist[t.x][t.y];

        for(int i = 0; i < 4; i++)
        {
            int xx = dx[i] + t.x , yy = dy[i] + t.y;
            int gx = ix[i] + t.x , gy = iy[i] + t.y; // gx,gy是找字符里的“/,\”这类的坐标

            if( xx < 0 || xx > n || yy <0 || yy > m )   continue;
            // (t.x,t.y) (xx,yy)分别是两个点，而中间的线(权值)就是w
            int w = (g[gx][gy] != dir[i]);//能联通表示 0,无需操作,否则为 1，表示需要而外操作
            if(dist[xx][yy] >= dist[t.x][t.y] + w)
            {
                dist[xx][yy] = dist[t.x][t.y] + w;
                if(w)   q.push_back({xx,yy}); //权值大(1)加到队尾
                else    q.push_front({xx,yy}); //权值小(0)加到队头
            }
        }
    }

    return -1;
```

```
}

int main()
{
    int T;
    cin >> T;
    while(T --)
    {
        cin >> n >> m;
        for(int i = 0; i < n; i++)  cin >> g[i];

        if( (n + m) & 1 )   puts("NO SOLUTION"); //奇点，无法从起点(0,0)沿对角线到达，例如(1,2)
        else cout << bfs() << endl;
    }

    return 0;
}
```

双向广搜 BFS

从起点和终点两个方向开始搜索，能降低枚举次数(BFS入队数量成指数级增长，两边往中间搜能降低很多)

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

const int N = 6;
int n;
string a[N],b[N];

int extend(queue<string>& q,unordered_map<string,int> &
da,unordered_map<string,int>& db,string a[],string b[])
{
    string t = q.front();
    q.pop();

    for(int j = 0; j < n; j++)
    {
        for(int i = 0; i < t.size(); i++)
            if( i + a[j].size() <= t.size() && t.substr(i,a[j].size()) == a[j])
            {
                string state = t.substr(0,i) + b[j] + t.substr(i + a[j].size());

                if(da.count(state)) continue;
                if(db.count(state)) return da[t] + 1 + db[state];

                da[state] = da[t] + 1;
                q.push(state);
            }
    }
```

```
    return 11;
}

int bfs(string A,string B)
{
    queue<string> qa;
    queue<string> qb;
    unordered_map<string,int> da;
    unordered_map<string,int> db;

    qa.push(A),qb.push(B);
    da[A] = 0,db[B] = 0;

    while(qa.size() && qb.size())
    {
        int t;

        if(qa.size() <= qb.size())
            t = extend(qa,da,db,a,b);
        else
            t = extend(qb,db,da,b,a);

        if(t <= 10) return t;
    }

    return 11;
}

int main()
{
    string A,B;
    cin >> A >> B;

    while(cin >> a[n] >> b[n++]) ;

    int t = bfs(A,B);

    if(t > 10)  puts("NO ANSWER!");
    else    cout << t << endl;

    return 0;
}
```

A star 启发式搜索

A star算法是一直沿着某条估计出来的路劲搜索的，且估计出来的路劲一定要小于等于实际路劲

A star的形式与Dijkstra的形式非常接近（Dijkstra估计函数为0），但是Dijkstra在每次出队之后就已经定下来出队点已是最小距离，而A star不行。A star只能保证终点状态是最小的。

八码数 题解

AC代码

```
```

```cpp
#include<iostream>
#include<algorithm>
#include<unordered_map>
#include<queue>
#include<string>
using namespace std;

int dx[] = {-1,0,1,0};
int dy[] = {0,1,0,-1};
char op[5] = "urdl";

int f(string state)
{//曼哈顿估价函数，估价值为每个位置离终点的曼哈顿距离之和
    int res = 0;
    for(int i = 0; i < 9; i++)
        if(state[i] != 'x')
        {
            int t = state[i] - '1';
            res += abs(t/3 - i/3) + abs(t%3 - i%3);
        }

    return res;
}

void bfs(string start)
{
    string end = "12345678x";

    unordered_map<string,int> dist; //存真实距离
    unordered_map<string,pair<char,string>>pre; //存路径
    priority_queue<pair<int,string>,vector<pair<int,string>
>,greater<pair<int,string> > > heap;//存离终点的估计距离
    dist[start] = 0;
    heap.push({dist[start] + f(start),start});

    while(heap.size())
    {
        auto t = heap.top();
        heap.pop();

        string state = t.second , source = t.second;
        int distance = t.first;

        if(state == end)
        { //到达终点,输出路径
            string path;
            while(start != end)
            {
                path += pre[end].first;
                end = pre[end].second;
            }
            reverse(path.begin(),path.end());
            cout << path << endl;
            return;
        }

        int location;
        for(int i = 0; i < 9; i++)
```

```
            if(state[i] == 'x') location = i;//找到'x'的位置

        for(int i = 0; i < 4; i++)
        {//向四个方向扩展
            state = source;
            int x = (location / 3) + dx[i];
            int y = (location % 3) + dy[i];

            swap(state[location],state[x*3 + y]);

            if(x < 0 || x >= 3 || y < 0 || y >= 3)  continue;
            if(dist.count(state) == 0 || dist[state] > dist[source] + 1)
            { // 如果是第一次扩展到这个状态 或 走到这个状态有更小的步数
                dist[state] = dist[source] + 1;
                heap.push({dist[state] + f(state),state});
                pre[state] = {op[i],source};
            }
        }
    }

}

int main()
{
    string start;

    for(int i = 0; i < 9;i++)
    {
        char c;
        cin >> c;
        start += c;
    }

    int cnt = 0;
    for(int i = 0; i < 9; i++)
        for(int j = i; j < 9; j++)
            if(start[i] != 'x' && start[j] != 'x' && start[i] > start[j])
                cnt++;

    if(cnt & 1) puts("unsolvable");
    else bfs(start);

    return 0;
}
```

[第k短路]

AC代码

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

const int N = 1010, M = 1e5 + 10;
```

```
int n, m, S, T, K, dist[N];
int h[N], rear_h[N], w[M], e[M], ne[M], idx;
bool st[N];

void add(int a, int b, int c, int h[])
{
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
}

void Dijkstra(int s)
{
    memset(dist, 0x3f, sizeof dist);
    priority_queue<pair<int, int>, vector<pair<int, int> >, greater<pair<int,
int> > >heap;
    heap.push({ 0,s });
    dist[s] = 0;

    while (heap.size())
    {
        auto t = heap.top();
        heap.pop();

        int ver = t.second;

        if (st[ver])    continue;
        st[ver] = true;

        for (int i = rear_h[ver]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dist[ver] + w[i])
            {
                dist[j] = dist[ver] + w[i];
                heap.push({ dist[j],j });
            }
        }
    }
}

int a_star_bfs(int s, int end)
{
    int cnt = 0;
    priority_queue<pair<int, int>, vector<pair<int, int> >, greater<pair<int,
int> > >heap;
    if (dist[s] == 0x3f3f3f3f)  return -1;
    heap.push({ dist[s],s });

    while (heap.size())
    {
        auto t = heap.top();
        heap.pop();

        int ver = t.second, t_dist = t.first - dist[ver];

        if (ver == end) cnt++;
        if (cnt == K)   return t.first;
```

```
        for (int i = h[ver]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] == 0x3f3f3f3f)  continue;
            heap.push({ dist[j] + t_dist + w[i],j });
        }
    }

    return -1;
}

int main()
{
    memset(h, -1, sizeof h);
    memset(rear_h, -1, sizeof rear_h);

    for (cin >> n >> m; m; m--)
    {
        int a, b, w;
        cin >> a >> b >> w;
        add(a, b, w, h);
        add(b, a, w, rear_h);
    }

    cin >> S >> T >> K;

    if (S == T) K++;

    Dijkstra(T);

    cout << a_star_bfs(S, T) << endl;

    return 0;
}
```