**3.1.3 BFS的代码实现**

BFS（**静态版**二叉树）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | #include <bits/stdc++.h>  using namespace std;  const int N = 100005;  struct Node{ //用静态数组记录二叉树  char value;  int lson, rson; //左右孩子  }tree[N]; //tree[0]不用，0表示空结点  int index = 1; //记录结点存在tree[]的位置，从tree[1]开始用  int newNode(char val){  tree[index].value = val;  tree[index].lson = 0; //0表示空，tree[0]不用  tree[index].rson = 0;  return index ++;  }  void Insert(int &father, int child, int l\_r){ //插入孩子  if(l\_r == 0) tree[father].lson = child; //左孩子  else tree[father].rson = child; //右孩子  }  int buildtree(){ //建一棵二叉树  int A = newNode('A');int B = newNode('B');int C = newNode('C');  int D = newNode('D');int E = newNode('E');int F = newNode('F');  int G = newNode('G');int H = newNode('H');int I = newNode('I');  Insert(E,B,0); Insert(E,G,1); //E的左孩子是B，右孩子是G  Insert(B,A,0); Insert(B,D,1);  Insert(G,F,0); Insert(G,I,1);  Insert(D,C,0); Insert(I,H,0);  int root = E;  return root;  }  int main(){  int root = buildtree();  queue <int> q;  q.push(root); //从根结点开始  while(q.size()){  int tmp = q.front();  cout << tree[tmp].value << " "; //打印队头  q.pop(); //去掉队头  if(tree[tmp].lson != 0) q.push(tree[tmp].lson); //左孩子入队  if(tree[tmp].rson != 0) q.push(tree[tmp].rson); //右孩子入队  }  return 0;  } |

BFS（指针版二叉树）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | #include <bits/stdc++.h>  using namespace std;  struct node{ //指针二叉树  char value;  node \*l, \*r;  node(char value='#',node \*l=NULL, node \*r=NULL):value(value), l(l), r(r){}  };  void remove\_tree(node \*root){ //释放空间  if(root == NULL) return;  remove\_tree(root->l);  remove\_tree(root->r);  delete root;  }  int main(){  node \*A,\*B,\*C,\*D,\*E,\*F,\*G,\*H,\*I; //以下建一棵二叉树  A = new node('A'); B = new node('B'); C = new node('C');  D = new node('D'); E = new node('E'); F = new node('F');  G = new node('G'); H = new node('H'); I = new node('I');  E->l = B; E->r = G; B->l = A; B->r = D;  G->l = F; G->r = I; D->l = C; I->l = H; //以上建了一棵二叉树  queue <node> q;  q.push(\*E);  while(q.size()){  node \*tmp;  tmp = &(q.front());  cout << tmp->value << " "; //打印队头  q.pop(); //去掉队头  if(tmp->l) q.push(\*(tmp->l)); //左孩子入队  if(tmp->r) q.push(\*(tmp->r)); //右孩子入队  }  remove\_tree(E);  return 0;  } |

**3.1.4 DFS的常见操作和代码框架**

DFS（静态数组版二叉树）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114 | #include <bits/stdc++.h>  using namespace std;  const int N = 100005;  struct Node{  char value; int lson, rson;  }tree[N]; //tree[0]不用，0表示空结点  int index = 1; //记录结点存在tree[]的位置，从tree[1]开始用  int newNode(char val){ //新建结点  tree[index].value = val;  tree[index].lson = 0; //0表示空，tree[0]不用  tree[index].rson = 0;  return index ++;  }  void insert(int &father, int child, int l\_r){ //插入孩子  if(l\_r == 0) tree[father].lson = child; //左孩子  else tree[father].rson = child; //右孩子  }  int dfn[N] = {0}; //dfn[i]是结点i的时间戳  int dfn\_timer = 0;  void dfn\_order (int father){  if(father != 0){  dfn[father] = ++dfn\_timer;  printf("dfn[%c]=%d; ", tree[father].value, dfn[father]);//打印时间戳  dfn\_order (tree[father].lson);  dfn\_order (tree[father].rson);  }  }  int visit\_timer = 0;  void visit\_order (int father){ //打印DFS序  if(father != 0){  printf("visit[%c]=%d; ", tree[father].value, ++visit\_timer);  //打印DFS序：第1次访问结点  visit\_order (tree[father].lson);  visit\_order (tree[father].rson);  printf("visit[%c]=%d; ", tree[father].value, ++visit\_timer);  //打印DFS序：第2次回溯  }  }  int deep[N] = {0}; //deep[i]是结点i的深度  int deep\_timer = 0;  void deep\_node (int father){  if(father != 0){  deep[father] = ++deep\_timer; //打印树的深度，第一次访问时，深度+1  printf("deep[%c]=%d; ",tree[father].value,deep[father]);  deep\_node (tree[father].lson);  deep\_node (tree[father].rson);  deep\_timer--; //回溯时，深度-1  }  }  int num[N] = {0}; //num[i]是以i为父亲的子树上的结点总数  int num\_node (int father){  if(father == 0) return 0;  else{  num[father] = num\_node (tree[father].lson) +  num\_node (tree[father].rson) + 1;  printf("num[%c]=%d; ", tree[father].value, num[father]); //打印数量  return num[father];  }  }  void preorder (int father){ //求先序序列  if(father != 0){  cout << tree[father].value <<" "; //先序输出  preorder (tree[father].lson);  preorder (tree[father].rson);  }  }  void inorder (int father){ //求中序序列  if(father != 0){  inorder (tree[father].lson);  cout << tree[father].value <<" "; //中序输出  inorder (tree[father].rson);  }  }  void postorder (int father){ //求后序序列  if(father != 0){  postorder (tree[father].lson);  postorder (tree[father].rson);  cout << tree[father].value <<" "; //后序输出  }  }  int buildtree(){ //建一棵树  int A = newNode('A');int B = newNode('B');int C = newNode('C'); //定义结点  int D = newNode('D');int E = newNode('E');int F = newNode('F');  int G = newNode('G');int H = newNode('H');int I = newNode('I');  insert(E,B,0); insert(E,G,1); //建树。E的左孩子是B，右孩子是G  insert(B,A,0); insert(B,D,1); insert(G,F,0); insert(G,I,1);  insert(D,C,0); insert(I,H,0);  int root = E;  return root;  }  int main(){  int root = buildtree();  cout <<"dfn order: "; dfn\_order(root); cout << endl; //打印时间戳  cout <<"visit order: "; visit\_order(root); cout << endl; //打印DFS序  cout <<"deep order: "; deep\_node(root); cout << endl; //打印结点深度  cout <<"num of tree: "; num\_node(root); cout << endl; //打印子树上的结点数  cout <<"in order: "; inorder(root); cout << endl; //打印中序序列  cout <<"pre order: "; preorder(root); cout << endl; //打印先序序列  cout <<"post order: "; postorder(root); cout << endl; //打印后序序列  return 0;  }  /\* 输出是：  dfn order: dfn[E]=1; dfn[B]=2; dfn[A]=3; dfn[D]=4; dfn[C]=5; dfn[G]=6; dfn[F]=7; dfn[I]=8; dfn[H]=9;  visit order: visit[E]=1; visit[B]=2; visit[A]=3; visit[A]=4; visit[D]=5; visit[C]=6; visit[C]=7; visit[D]=8; visit[B]=9; visit[G]=10; visit[F]=11; visit[F]=12; visit[I]=13; visit[H]=14; visit[H]=15; visit[I]=16; visit[G]=17; visit[E]=18;  deep order: deep[E]=1; deep[B]=2; deep[A]=3; deep[D]=3; deep[C]=4; deep[G]=2; deep[F]=3; deep[I]=3; deep[H]=4;  num of tree: num[A]=1; num[C]=1; num[D]=2; num[B]=4; num[F]=1; num[H]=1; num[I]=2; num[G]=4; num[E]=9;  in order: A B C D E F G H I  pre order: E B A D C G F I H  post order: A C D B F H I G E \*/ |

DFS（指针版二叉树）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | #include <bits/stdc++.h>  using namespace std;  struct node{  char value;  node \*l, \*r;  node(char value = '#', node \*l = NULL, node \*r = NULL):value(value), l(l), r(r){}  };  void preorder (node \*root){ //求先序序列  if(root != NULL){  cout << root->value <<" "; //先序输出  preorder (root ->l);  preorder (root ->r);  }  }  void inorder (node \*root){ //求中序序列  if(root != NULL){  inorder (root ->l);  cout << root->value <<" "; //中序输出  inorder (root ->r);  }  }  void postorder (node \*root){ //求后序序列  if(root != NULL){  postorder (root ->l);  postorder (root ->r);  cout << root->value <<" "; //后序输出  }  }  void remove\_tree(node \*root){ //释放空间  if(root == NULL) return;  remove\_tree(root->l);  remove\_tree(root->r);  delete root;  }  int main(){  node \*A, \*B,\*C,\*D,\*E,\*F,\*G,\*H,\*I;  A = new node('A'); B = new node('B'); C = new node('C');  D = new node('D'); E = new node('E'); F = new node('F');  G = new node('G'); H = new node('H'); I = new node('I');  E->l = B; E->r = G; B->l = A; B->r = D;  G->l = F; G->r = I; D->l = C; I->l = H;  cout <<"in order: "; inorder(E); cout << endl; //打印中序序列  cout <<"pre order: "; preorder(E); cout << endl; //打印先序序列  cout <<"post order: "; postorder(E); cout << endl; //打印后序序列  remove\_tree(E);  return 0;  } |

**3.1.6 连通性判断**

|  |
| --- |
| 例3-1. 全球变暖https://www.lanqiao.cn/problems/178/learning/ |

**1. DFS求解连通性问题**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | #include<bits/stdc++.h>  using namespace std;  const int N = 1010;  char mp[N][N]; //地图  int vis[N][N]={0}; //标记是否搜过  int d[4][2] = {{0,1}, {0,-1}, {1,0}, {-1,0}}; //四个方向  int flag; //用于标记这个岛中是否被完全淹没  void dfs(int x, int y){  vis[x][y] = 1; //标记这个'#'被搜过。注意为什么放在这里  if( mp[x][y+1]=='#' && mp[x][y-1]=='#' &&  mp[x+1][y]=='#' && mp[x-1][y]=='#' )  flag = 1; //上下左右都是陆地，这是一个高地，不会淹没  for(int i = 0; i < 4; i++){ //继续DFS周围的陆地  int nx = x + d[i][0], ny = y + d[i][1];  if(vis[nx][ny]==0 && mp[nx][ny]=='#') //注意为什么要判断vis[][]  //继续DFS未搜过的陆地，目的是标记它们  dfs(nx,ny);  }  }  int main(){  int n; cin >> n;  for (int i = 0; i < n; i++) cin >> mp[i];  int ans = 0 ;  for(int i = 1; i <= n; i++) //DFS所有像素点  for(int j = 1; j <= n; j++)  if(mp[i][j]=='#' && vis[i][j]==0){  flag = 0; //假设这个岛被淹  dfs(i,j); //找这个岛中有没有高地，如果有,置flag=1  if(flag == 0) ans++; //这个岛被淹了，统计被淹没岛的数量  }  cout<<ans<<endl;  return 0;  } |

**2. BFS求解连通性问题**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | #include<bits/stdc++.h>  using namespace std;  const int N = 1010;  char mp[N][N];  int vis[N][N];  int d[4][2] = {{0,1}, {0,-1}, {1,0}, {-1,0}}; //四个方向  int flag;  void bfs(int x, int y) {  queue<pair<int, int>> q;  q.push({x, y});  vis[x][y] = 1; //标记这个'#'被搜过  while (q.size()) {  pair<int, int> t = q.front();  q.pop();  int tx = t.first, ty = t.second;  if( mp[tx][ty+1]=='#' && mp[tx][ty-1]=='#' &&  mp[tx+1][ty]=='#' && mp[tx-1][ty]=='#' )  flag = 1; //上下左右都是陆地，不会淹没  for (int i = 0; i < 4; i++) { //扩展(tx,ty)的4个邻居  int nx = tx + d[i][0], ny = ty + d[i][1];  if(vis[nx][ny]==0 && mp[nx][ny]=='#'){ //把陆地放进队列  vis[nx][ny] = 1; //注意：这一句必不可少  q.push({nx, ny});  }  }  }  }  int main() {  int n; cin >> n;  for (int i = 0; i < n; i++) cin >> mp[i];  int ans = 0;  for (int i = 0; i < n; i++)  for (int j = 0; j < n; j++)  if (mp[i][j] == '#' && vis[i][j]==0) {  flag = 0;  bfs(i, j);  if(flag == 0) ans++; //这个岛全部被淹，统计岛的数量  }  cout << ans << endl;  return 0;  } |

**3.2.1 BFS判重**

|  |  |
| --- | --- |
| 例3-2. 跳蚱蜢https://www.lanqiao.cn/problems/642/learning/ | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | #include<bits/stdc++.h>  using namespace std;  struct node{  node(){}  node(string ss, int tt){s = ss, t = tt;}  string s;  int t;  };  //(1) map  map<string, bool> mp;  //(2) set  // set<string> visited; //记录已经搜索过的状态  queue<node> q;  void solve(){  while(!q.empty()){  node now = q.front();  q.pop();  string s = now.s;  int step = now.t;  if(s == "087654321"){ cout<<step<<endl; break;} //到目标了，输出跳跃步数  int i;  for(i = 0 ; i < 10 ; i++) //找到盘子的位置i  if(s[i] == '0') break;  for(int j = i - 2 ; j <= i + 2 ; j++){ //4种跳法  int k = (j + 9) % 9;  if(k == i) continue; //这是当前状态，不用检查  string news = s;  char tmp = news[i];  news[i] = news[k];  news[k] = tmp; //跳到一种情况  //(1) map  if(!mp[news]){ //判重：这个情况没有出现过  mp[news] = true;  q.push(node(news, step + 1));  }  //(2)set  /\* if(visited.count(news)==0){ //判重：这个情况没有出现过  visited.insert(news);  q.push(node(news, step + 1));  } \*/  }  }  }  int main(){  string s = "012345678";  q.push(node(s, 0));  //(1) map  mp[s] = true;  solve();  return 0;  } |

**3.2.2 剪枝应用**

|  |
| --- |
| 例3-6. Tempter of the Bone hdu 1010 |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | //改写自：https://www.cnblogs.com/CSU3901130321/p/3993740.html  #include <bits/stdc++.h>  using namespace std;  char mat[8][8],visit[8][8];  int n, m, t;  int flag; //flag=1，表示找到了答案  int a, b, c, d; //起点S(a,b)，终点D(c,d)  int dir[4][2] = {{1,0}, {-1,0}, {0,1}, {0,-1}}; //上下左右4个方向  #define CHECK(xx,yy) (xx>=0 && xx<n && yy>=0 && yy<m) //是否在迷宫中  void dfs(int x, int y, int time){  if(flag) return; //逐层退出DFS，有多少层DFS，就退多少次  if(mat[x][y] == 'D'){  if(time == t) flag = 1; //找到答案  return; //D只能走一次，所以不管对不对，都返回  }  //if(time > t) return; //剪枝(1)：因为有剪枝(2)，(1)就多余了  int tmp = t - time - abs(c-x) - abs(d-y);  if(tmp < 0) return; //剪枝（2）  //if(tmp & 1) return; //奇偶剪枝：不应该在这里做，应该在main里做  for(int i=0; i<4; i++){ //上下左右  int xx = x + dir[i][0], yy = y + dir[i][1];  if(CHECK(xx,yy) && mat[xx][yy]!='X' && !visit[xx][yy]){  visit[xx][yy] = 1; //地板标记为走过，不能再走  dfs(xx, yy, time + 1); //遍历所有的路径  visit[xx][yy] = 0; //递归返回，这块地板恢复为没走过  }  }  return;  }  int main(){  while(~scanf("%d%d%d",&n,&m,&t)){  if(n==0 && m==0 && t==0) break;  for(int i=0;i<n;i++)  for(int j=0;j<m;j++){  cin>>mat[i][j];  if(mat[i][j] == 'S') a=i,b=j;  if(mat[i][j] == 'D') c=i,d=j;  }  memset(visit, 0, sizeof(visit));  int tmp = t - abs(c-a) - abs(d-b); //在DFS之前，做奇偶判断  if(tmp & 1){ puts("NO"); continue; } //无解，不用DFS了  flag = 0;  visit[a][b] = 1; //标记起点已经走过  dfs(a, b, 0); //搜索路径  if(flag) puts("YES");  else puts("NO");  }  return 0;  } |

3.4 BFS与最短路

|  |  |
| --- | --- |
| 例3-15. 迷宫https://www.lanqiao.cn/problems/602/learning/ | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61 | #include<bits/stdc++.h>  using namespace std;  struct node{  int x;  int y;  //(1)简单方法：  string path; //path,记录从起点(0,0)到这个点(x,y)的完整路径  };  char mp[31][51]; //存地图  char k[4]={'D','L','R','U'}; //字典序  int dir[4][2]={{1,0},{0,-1},{0,1},{-1,0}};  int vis[30][50]; //标记。vis=1: 已经搜过，不用再搜  //(2)标准方法：  char pre[31][51]; // 用于查找前驱点。例如pre[x][y] = ‘D’，表示上一个点  //往下走一步到了(x,y)，那么上一个点是(x-1,y)  void print\_path(int x,int y){ //打印路径：从(0,0)到(29,49)  if(x==0 && y==0) return; //回溯到了起点，递归结束，返回  if(pre[x][y]=='D') print\_path(x-1,y); //回溯，往上 U  if(pre[x][y]=='L') print\_path(x, y+1); //回溯，往右 R  if(pre[x][y]=='R') print\_path(x, y-1);  if(pre[x][y]=='U') print\_path(x+1,y);  printf("%c",pre[x][y]); //最后打印的是终点  }  void bfs(){  node start; start.x=0; start.y=0;  //(1)简单方法：  start.path="";  vis[0][0]=1; //标记起点被搜过  queue<node>q;  q.push(start); //把第一个点放进队列，开始BFS  while(!q.empty()){  node now = q.front(); //取出队首  q.pop();  if(now.x==29 && now.y==49){ //第一次达到终点，这就是字典序最小的最短路径  //(1)简单方法：打印完整路径  cout << now.path << endl;  //(2)标准方法：打印完整路径，从终点回溯到起点，打印出来是从起点到终点的正序  print\_path(29,49);  return;  }  for(int i=0;i<4;i++){ //扩散邻居结点  node next;  next.x = now.x + dir[i][0]; next.y = now.y + dir[i][1];  if(next.x<0||next.x>=30||next.y<0||next.y>=50) //越界了  continue;  if(vis[next.x][next.y]==1 || mp[next.x][next.y]=='1')  continue; //vis=1:已经搜过; mp=1:是障碍  vis[next.x][next.y]=1; //标记被搜过  //(1)简单方法：记录完整路径：复制上一个点的路径，加上这一步  next.path = now.path + k[i];  //(2)标准方法：记录点(x,y)的前驱  pre[next.x][next.y] = k[i];  q.push(next);  }  }  }  int main(){  for(int i=0;i<30;i++) cin >> mp[i]; //读题目给的地图数据  bfs();  } | |

3.6 BFS与优先队列

|  |
| --- |
| 例3-20. 最短路 https://www.lanqiao.cn/problems/1122/learning/ |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62 | #include<bits/stdc++.h>  using namespace std;  const long long INF = 0x3f3f3f3f3f3f3f3fLL; //这样定义的好处是: INF <= INF+x  const int N = 3e5+2;  struct edge{  int from, to; //边：起点，终点，权值。起点from并没有用到，e[i]的i就是from  long long w; //边：权值  edge(int a, int b,long long c){from=a; to=b; w=c;}  };  vector<edge>e[N]; //存储图  struct node{  int id; long long n\_dis; //id：结点；n\_dis：这个结点到起点的距离  node(int b,long long c){id=b; n\_dis=c;}  bool operator < (const node & a) const  { return n\_dis > a.n\_dis;}  };  int n,m;  int pre[N]; //记录前驱结点  void print\_path(int s, int t) { //打印从s到t的最短路  if(s==t){ printf("%d ", s); return; } //打印起点  print\_path(s, pre[t]); //先打印前一个点  printf("%d ", t); //后打印当前点。最后打印的是终点t  }  long long dis[N]; //记录所有结点到起点的距离  bool done[N]; //done[i]=true表示到结点i的最短路径已经找到  void dijkstra(){  int s = 1; //起点s = 1  for (int i=1;i<=n;i++) {dis[i]=INF; done[i]=false; } //初始化  dis[s]=0; //起点到自己的距离是0  priority\_queue <node> Q; //优先队列，存结点信息  Q.push(node(s, dis[s])); //起点进队列  while (!Q.empty()) {  node u = Q.top(); //pop出距起点s距离最小的结点u  Q.pop();  if(done[u.id]) continue; //丢弃已经找到最短路径的结点。即集合A中的结点  done[u.id]= true;  for (int i=0; i<e[u.id].size(); i++) { //检查结点u的所有邻居  edge y = e[u.id][i]; //u.id的第i个邻居是y.to  if(done[y.to]) continue; //丢弃已经找到最短路径的邻居结点  if (dis[y.to] > y.w + u.n\_dis) {  dis[y.to] = y.w + u.n\_dis;  Q.push(node(y.to, dis[y.to])); //扩展新邻居，放到优先队列中  pre[y.to]=u.id; //如果有需要，记录路径  }  }  }  // print\_path(s,n); //如果有需要，打印路径: 起点1，终点n  }  int main(){  scanf("%d%d",&n,&m);  for (int i=1;i<=n;i++) e[i].clear();  while (m--) {  int u,v,w; scanf("%d%d%lld",&u,&v,&w);  e[u].push\_back(edge(u,v,w));  // e[v].push\_back(edge(v,u,w)); //本题是单向边  }  dijkstra();  for(int i=1;i<=n;i++){  if(dis[i]>=INF) cout<<"-1 ";  else printf("%lld ", dis[i]);  }  } |

3.7 BFS与双端队列

|  |  |
| --- | --- |
| 例3-21. Switch the Lamp On 洛谷 P4667 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | #include<bits/stdc++.h>  using namespace std;  const int dir[4][2] = {{-1,-1},{-1,1},{1,-1},{1,1}}; //4个方向的位移  const int ab[4] = {2,1,1,2}; //4个元件期望的方向  const int cd[4][2] = {{-1,-1},{-1,0},{0,-1},{0,0}}; //4个元件编号的位移  int graph[505][505],dis[505][505]; //dis记录结点到起点s的最短路  struct P{ int x,y,dis; }u;  int read\_ch(){  char c;  while((c = getchar())!='/' && c != '\\') ; //字符不是'/'和'\'  return c=='/' ? 1 : 2;  }  int main(){  int n, m; cin >>n >>m;  memset(dis,0x3f,sizeof(dis));  for(int i=1;i<=n;++i)  for(int j=1;j<=m;++j) graph[i][j] = read\_ch();  deque <P> dq;  dq.push\_back((P){1,1,0});  dis[1][1] = 0;  while(!dq.empty()){  u = dq.front(), dq.pop\_front(); //front()读队头,pop\_front()弹出队头  int nx,ny;  for(int i=0;i<=3;++i) { //4个方向  nx = u.x+dir[i][0]; ny = u.y+dir[i][1];  int d = 0; //边权  d = graph[u.x+cd[i][0]][u.y+cd[i][1]]!=ab[i]; //若方向不相等,则d=1  if(nx && ny && nx<n+2 && ny<m+2 && dis[nx][ny]>dis[u.x][u.y]+d){  // 如果一个结点再次进队，那么距离应该更小。实际上，  //由于再次进队时，距离肯定更大，所以这里的作用是阻止再次入队  dis[nx][ny] = dis[u.x][u.y]+d;  if(d==0) dq.push\_front((P){nx, ny, dis[nx][ny]}); //边权=0，插到队头  else dq.push\_back ((P){nx, ny, dis[nx][ny]}); //边权=1，插到队尾  if(nx==n+1 && ny==m+1) break; //到终点退出。不退也行，队列空自动退  }  }  }  if(dis[n+1][m+1] != 0x3f3f3f3f) cout << dis[n+1][m+1];  else cout <<"NO SOLUTION"; //可能无解，即s到t不通  return 0;  } |

**3.9.2 IDA\***

|  |
| --- |
| 例3-24. poj 3134 - Power Calculus |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | #include<stdio.h>  #include<string.h>  const int N = 100; //最大层次  int num[N]; //记录一条路径上的数字，num[i]是路径上第i层的数字  int n, depth;  bool dfs(int now, int d) { //now:当前路径走到的数字，d：now所在的深度  if (d > depth) return false; //当前深度大于层数限制  if (now == n) return true; //找到目标。注意：这一句不能放在上一句前面  if (now << (depth - d) < n) //剪枝：剩下的层数用最乐观的倍增也不能达到n  return false;  num[d] = now; //记录这条路径上第d层的数字  for(int i = 0; i <= d; i++) { //遍历之前算过的数，继续下一层  if (dfs(now + num[i], d + 1)) return true; //加  else if (dfs(now - num[i], d + 1)) return true; //减  }  return false;  }  int main() {  while(~scanf("%d", &n) && n) {  for(depth = 0;; depth++) { //IDDFS：每次限制最大搜索depth层  memset(num, 0, sizeof(num));  if (dfs(1, 0)) break; //从数字1开始，当前层0  }  printf("%d\n", depth);  }  return 0;  } |