第3章 搜索

搜索包括宽度优先搜索和深度优先搜索，这两种算法是算法竞赛的基础。本节全面介绍这两种算法的思想、编码、应用和扩展。它们不仅能直接用于解决问题，也启发了很多高级算法。本章是全书的基础，希望读者能彻底学懂学通，再继续学习后面的章节。

扫一扫

视频讲解

3.1 BFS 和 DFS 基础

3.1.1 搜索简介

搜索是“暴力法”算法思想的具体实现。暴力法(Brute Force)又称为蛮力法,把所有可能的情况都罗列出来,然后逐一检查,从中找到答案。这种方法简单直接,不玩花样,利用了计算机强大的计算能力。

搜索是“通用”的方法。如果一个问题比较难,可以先尝试搜索,或许能启发出更好的算法。竞赛时遇到难题,如果有时间,试试用搜索提交,说不定判题数据很弱,就通过了。

搜索的思路很简单,但是操作起来也并不容易,一般有以下操作步骤。

(1) 找到所有可能的数据,并且用数据结构表示和存储。

(2) 优化。尽量多地排除不符合条件的数据,以减少搜索的空间。

(3) 用某个算法快速检索这些数据。

3.1.2 搜索算法的基本思路

搜索的基本算法分为两种:宽度优先搜索(Breadth-First Search,BFS)(或称为广度优先搜索)和深度优先搜索(Depth-First Search,DFS)。

这两个算法的思想可以用“老鼠走迷宫”的例子说明,又形象又透彻。迷宫内部的路错综复杂,老鼠从入口进去后,怎样才能找到出口?有两种不同的方法。

(1) 一只老鼠走迷宫。它在每个路口都选择先走左边(先走右边也可以),能走多远就走多远,直到碰壁无法继续前进,然后回退一步并改走右边,继续往下走。用这个办法能走遍所有路,而且不会重复(回退不算重复)。这个思路就是 DFS。

(2) 一群老鼠走迷宫。假设老鼠无限多,这群老鼠进入迷宫后,在每个路口都派出部分老鼠探索所有没走过的路。走某条路的老鼠,如果碰壁无法前行就停下;如果到达的路口已经有别的老鼠探索过了,也停下。显然,所有道路都能走到,而且不会重复。这个思路就是 BFS。BFS 看起来像“并行计算”,不过由于程序是单机顺序运行的,可以把 BFS 看作并行计算的模拟。

BFS 是“全面扩散、逐层递进”;DFS 是“一路到底、逐步回退”。

下面以如图 3.1 所示的一棵二叉树为例,演示 BFS 和 DFS 的访问顺序。

BFS 的访问顺序是{*E B G A D F I C H*},即第 1 层(*E*)→第 2 层(*BG*)→第 3 层(*ADFI*)→第 4 层(*CH*)。BFS 的特点是分层访问,每层访问完毕后才访问下一层。

DFS 的访问顺序是先访问左节点再访问右节点,访问顺序是{*E B A D C G F I H*}。需要注意的是,访问顺序不是输出顺序。例如,上面的二叉树,它的 DFS 中序遍历、先序遍历、后序遍历都不同,但是对节点的访问顺序是一样的(实际上就是先序遍历)。

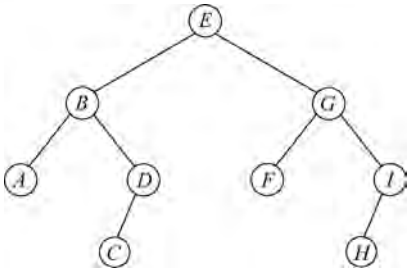


图 3.1　一棵二叉树

3.1.3　BFS 的代码实现

BFS 的代码实现可描述为“BFS＝队列”。为什么“BFS＝队列”？以老鼠走迷宫为例，从起点 s 开始一层一层地扩散出去，处理完离 s 近的第 i 层之后，再处理第 $i+1$ 层。这一操作用队列最方便，处理第 i 层的节点 a 时，把 a 的第 $i+1$ 层的邻居放到队列尾部即可。队列内的节点有以下两个特征。

(1) 处理完第 i 层后，才会处理第 $i+1$ 层。

(2) 队列中在任意时刻最多有两层节点，其中第 i 层节点都在第 $i+1$ 层前面。

下面给出 BFS 遍历图 3.1 二叉树的代码，有静态版二叉树和指针版二叉树两种代码，竞赛中一般用静态版二叉树，不易出错。两段代码都使用 STL queue，输出都是 $\{E\ B\ G\ A\ D\ F\ I\ C\ H\}$。

1. 静态版二叉树

```
#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 表示空节点
int index = 1;                                  //记录节点存在 tree[]的位置,从 tree[1]开始用
int newNode(char val){
    tree[index].value = val;
    tree[index].lson = 0;                       //tree[0]不用,0 表示空节点
    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);                                          //从根节点开始
    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;
}
```

下面分析 BFS 遍历二叉树的复杂度。

时间复杂度：由于需要检查每条边，且只检查一次，时间复杂度为 $O(m)$，m 为边的数量。

空间复杂度：每个点只进出队列各一次，空间复杂度为 $O(n)$，n 为点的数量。

2. 指针版二叉树

作为静态版二叉树代码的对照，下面给出指针版二叉树代码。

```
#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;                      //以下将建一棵二叉树
    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;
}
```

BFS是逐层扩散的，非常符合在图上计算最短路径，先扩散到的节点离根节点更近。很多最短路径算法都是从BFS发展出来的。

3.1.4 DFS的常见操作和代码框架

1. DFS的常见操作

DFS的工作原理就是递归的过程，即“DFS＝递归”。

DFS的代码比BFS更简短一些。本节后面给出的两段代码分别基于静态数组版二叉树和指针版二叉树。它们输出了图3.1所示二叉树的各种DFS操作，有时间戳、DFS序、树深度、子树节点数，另外还给出了二叉树的中序输出、先序输出、后序输出。DFS访问节点有以下常见操作。

(1) 节点第1次被访问的时间戳。用dfn[i]表示节点i第1次被访问的时间戳，dfn_order()函数打印出了时间戳：

```
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
```

时间戳打印的结果就是下面的“先序输出”。

(2) DFS序。在递归时每个节点来回处理两次，即第1次访问(前进)和第2次回溯(回退)。visit_order()函数打印出了DFS序{$E\ B\ A\ A\ D\ C\ C\ D\ B\ G\ F\ F\ I\ H\ H\ I\ G\ E$}。这个序列有一个重要特征：每个节点出现两次，被这两次包围起来的是以它为父节点的一棵子树。例如，序列中的{$B\ A\ A\ D\ C\ C\ D\ B$}就是以B为父节点的一棵子树；又如{$I\ H\ H\ I$}，是以I为父节点的一棵子树。这个特征是递归操作产生的。

(3) 树的深度。从根节点向子树DFS，每个节点第1次被访问时，深度加1；从这个节点回溯时，深度减1。用deep[i]表示节点i的深度，deep_node()函数打印出了深度：

```
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
```

(4) 子树节点总数。用 num[i]表示以 i 为父亲的子树上的节点总数。例如,以 B 为父节点的子树共有 4 个节点{$A\ B\ C\ D$}。只需要一次简单的 DFS 就能完成,每棵子树节点的数量等于它的两个子树的数量相加,再加 1,即加它自己。num_node()函数做了计算并打印出了以每个节点为父亲的子树上的节点数量。

(5) 先序输出。preorder()函数按父节点、左子节点、右子节点的顺序打印,输出{$E\ B\ A\ D\ C\ G\ F\ I\ H$}。

(6) 中序输出。inorder()函数按左子节点、父节点、右子节点的顺序打印,输出{$A\ B\ C\ D\ E\ F\ G\ H\ I$}。

(7) 后序输出。postorder()函数按左子节点、右子节点、父节点的顺序打印,输出{$A\ C\ D\ B\ F\ H\ I\ G\ E$}。

如果已知树的先序、中序、后序,可以确定一棵树。“先序+中序”或“后序+中序”都能确定一棵树,但是“先序+后序”不能确定一棵树。请回顾 1.4.2 节“二叉树的遍历”的说明。

竞赛中一般用静态版二叉树写代码。作为对照,后面也给出指针版二叉树的代码。

1) 静态数组版二叉树

```
#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 表示空节点
int index = 1;                         //记录节点存在 tree[]的位置,从 tree[1]开始
int newNode(char val){                 //新建节点
    tree[index].value = val;
    tree[index].lson = 0;              //tree[0]不用,0 表示空节点
    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]是节点 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;
```

```
29  void visit_order (int father){                    //打印 DFS 序
30      if(father != 0){
31          printf("visit[ %c] = %d; ", tree[father].value, ++visit_timer);
32                                                    //打印 DFS 序：第 1 次访问节点
33          visit_order (tree[father].lson);
34          visit_order (tree[father].rson);
35          printf("visit[ %c] = %d; ", tree[father].value, ++visit_timer);
36                                                    //打印 DFS 序：第 2 次回溯
37      }
38  }
39  int deep[N] = {0};                                //deep[i]是节点 i 的深度
40  int deep_timer = 0;
41  void deep_node (int father){
42      if(father != 0){
43          deep[father] = ++deep_timer;              //打印树的深度,第 1 次访问时,深度加 1
44          printf("deep[ %c] = %d; ",tree[father].value,deep[father]);
45          deep_node (tree[father].lson);
46          deep_node (tree[father].rson);
47          deep_timer--;                             //回溯时,深度减 1
48      }
49  }
50  int num[N] = {0};                                 //num[i]是以 i 为父亲的子树上的节点总数
51  int num_node (int father){
52      if(father == 0)  return 0;
53      else{
54          num[father] = num_node (tree[father].lson) +
55                        num_node (tree[father].rson) + 1;
56          printf("num[ %c] = %d; ", tree[father].value, num[father]);     //打印数量
57          return num[father];
58      }
59  }
60  void preorder (int father){                       //求先序序列
61      if(father != 0){
62          cout << tree[father].value <<" ";         //先序输出
63          preorder (tree[father].lson);
64          preorder (tree[father].rson);
65      }
66  }
67  void inorder (int father){                        //求中序序列
68      if(father != 0){
69          inorder (tree[father].lson);
70          cout << tree[father].value <<" ";         //中序输出
71          inorder (tree[father].rson);
72      }
73  }
74  void postorder (int father){                      //求后序序列
75      if(father != 0){
76          postorder (tree[father].lson);
77          postorder (tree[father].rson);
78          cout << tree[father].value <<" ";         //后序输出
79      }
80  }
81  int buildtree(){                                  //建一棵树
```

```
82      int A = newNode('A');int B = newNode('B');int C = newNode('C'); //定义节点
83      int D = newNode('D');int E = newNode('E');int F = newNode('F');
84      int G = newNode('G');int H = newNode('H');int I = newNode('I');
85      insert(E,B,0);  insert(E,G,1);                  //建树.E的左孩子是B,右孩子是G
86      insert(B,A,0);  insert(B,D,1);  insert(G,F,0);  insert(G,I,1);
87      insert(D,C,0);  insert(I,H,0);
88      int root = E;
89      return root;
90  }
91  int main(){
92      int root = buildtree();
93      cout <<"dfn order: ";     dfn_order(root); cout << endl;     //打印时间戳
94      cout <<"visit order: "; visit_order(root); cout << endl;     //打印DFS序
95      cout <<"deep order: ";    deep_node(root); cout << endl;     //打印节点深度
96      cout <<"num of tree: ";    num_node(root); cout << endl;     //打印子树上的节点数
97      cout <<"in order:   ";      inorder(root); cout << endl;     //打印中序序列
98      cout <<"pre order:  ";     preorder(root); cout << endl;     //打印先序序列
99      cout <<"post order: ";    postorder(root); cout << endl;     //打印后序序列
100     return 0;
101 }
102 /*    输出是:
103 dfn order: dfn[E] = 1; dfn[B] = 2; dfn[A] = 3; dfn[D] = 4; dfn[C] = 5; dfn[G] = 6; dfn[F] = 7;
104 dfn[I] = 8; dfn[H] = 9;
105 visit order: visit[E] = 1; visit[B] = 2; visit[A] = 3; visit[A] = 4; visit[D] = 5; visit[C] = 6;
106 visit[C] = 7;visit[D] = 8; visit[B] = 9; visit[G] = 10; visit[F] = 11; visit[F] = 12;
107 visit[I] = 13; visit[H] = 14; visit[H] = 15; visit[I] = 16; visit[G] = 17; visit[E] = 18;
108 deep order: deep[E] = 1; deep[B] = 2; deep[A] = 3; deep[D] = 3; deep[C] = 4; deep[G] = 2;
109 deep[F] = 3; deep[I] = 3; deep[H] = 4;
110 num of tree: num[A] = 1; num[C] = 1; num[D] = 2; num[B] = 4; num[F] = 1; num[H] = 1; num[I] = 2;
111 num[G] = 4; num[E] = 9;
112 in order:   A B C D E F G H I
113 pre order:   E B A D C G F I H
114 post order: A C D B F H I G E     */
```

分析 DFS 遍历二叉树的复杂度，和 BFS 差不多，时间复杂度为 $O(m)$，空间复杂度为 $O(n)$，因为使用了长度为 n 的递归栈。

2）指针版二叉树

```
1   #include <bits/stdc++.h>
2   using namespace std;
3   struct node{
4       char value;
5       node *l, *r;
6       node(char value = '#', node *l = NULL, node *r = NULL):value(value), l(l), r(r){}
7   };
8   void preorder (node *root){                  //求先序序列
9       if(root != NULL){
```

```
10          cout << root->value <<" ";        //先序输出
11          preorder (root ->l);
12          preorder (root ->r);
13      }
14  }
15  void inorder (node * root){               //求中序序列
16      if(root != NULL){
17          inorder (root ->l);
18          cout << root->value <<" ";        //中序输出
19          inorder (root ->r);
20      }
21  }
22  void postorder (node * root){             //求后序序列
23      if(root != NULL){
24          postorder (root ->l);
25          postorder (root ->r);
26          cout << root->value <<" ";        //后序输出
27      }
28  }
29  void remove_tree(node * root){            //释放空间
30      if(root == NULL) return;
31      remove_tree(root->l);
32      remove_tree(root->r);
33      delete root;
34  }
35  int main(){
36      node   * A, * B, * C, * D, * E, * F, * G, * H, * I;
37      A = new node('A'); B = new node('B'); C = new node('C');
38      D = new node('D'); E = new node('E'); F = new node('F');
39      G = new node('G'); H = new node('H'); I = new node('I');
40      E->l = B; E->r = G;       B->l = A; B->r = D;
41      G->l = F; G->r = I;       D->l = C;       I->l = H;
42      cout <<"in order:   ";    inorder(E); cout << endl;    //打印中序序列
43      cout <<"pre order:  ";   preorder(E); cout << endl;    //打印先序序列
44      cout <<"post order: ";  postorder(E); cout << endl;    //打印后序序列
45      remove_tree(E);
46      return 0;
47  }
```

DFS 搜索的特征是一路深入，适合处理节点间的先后关系、连通性等，在图论中应用很广泛。

2. DFS 代码框架

DFS 的代码看起来简单，但是初学者在逻辑上会感到难以理解。下面给出 DFS 的框架，帮助初学者学习。后续 3.2 节的例题 hdu 1010 是非常符合这个框架的示例，请仔细分析该例题的代码。请读者在大量编码的基础上，再回头体会这个框架的作用。

```
ans;                                  //答案，用全局变量表示
void dfs(层数,其他参数){
    if (出局判断){                    //到达最底层，或者满足条件退出
```

```
        更新答案;                          //答案一般用全局变量表示
        return;                            //返回到上一层
    }
    (剪枝)                                 //在进一步 DFS 之前剪枝
    for (枚举下一层可能的情况)              //对每个情况继续 DFS
        if (used[i] == 0) {                //如果状态 i 没有用过,就可以进入下一层
            used[i] = 1;                   //标记状态 i,表示已经用过,在更底层不能再使用
            dfs(层数 + 1,其他参数);        //下一层
            used[i] = 0;                   //恢复状态,回溯时不影响上一层对这个状态的使用
            }
    return;                                //返回到上一层
}
```

3.1.5 BFS 和 DFS 的对比

1. 时间复杂度对比

大多数情况下,BFS 和 DFS 的时间复杂度差不多,因为它们都需要搜索整个空间。以图这种数据结构为例,图中的所有 n 个点和所有 m 条边在一般情况下都应该至少访问一次,所以复杂度一般大于 $O(n+m)$。有时点和边会计算多次,如计算图上两个点之间的最短路径,一条路径包含很多点和边,一个点或一条边可能属于不同的路径,需要计算多次,复杂度超过 $O(n+m)$。

2. 空间对比

BFS 使用的空间往往比 DFS 大。BFS 的主要问题是可能耗费巨大的空间。例如,一棵满二叉树,第 k 层有 2^k 个节点,用队列处理 BFS 时,每弹出一个节点,就进队两个节点;到第 k 层,队列中就有 2^k 个节点,如 $k=32$,$2^k \approx 40$ 亿,即 4GB 空间。使用双向广搜和迭代加深搜索,可以在一定程度上改善这一问题,具体参考本书后续内容。

3. 搜索能力对比

DFS 没有 BFS 的空间消耗问题,但是可能搜索大量无效的节点。仍以满二叉树为例,DFS 沿着左子树深入,然后逐步回退访问右边的子树。如果解在偏右的子树上,DFS 仍然需要全部检索完左边的子树,才能轮到右边。如果这棵树很深,那么就会搜索左边这些大量无效的节点。用迭代加深搜索可以改善这一问题。

总结 BFS 和 DFS 的特点,DFS 更适合找一个任意可行解,BFS 更适合找全局最优解。

4. 扩展和优化

在 BFS 和 DFS 的基础上,发展出了剪枝、记忆化(DFS)、双向广搜(BFS)、迭代加深搜索(DFS)、A* 算法(BFS)、IDA* (DFS)等技术,大大提高了搜索的能力。

DFS 的代码比 BFS 更简单，如果一道题目用 BFS 和 DFS 都可以解，一般用 DFS。

BFS 常用的技巧是去重。例如，BFS 的队列，把状态放入队列时，需要判断这个状态是否已经在队列中处理过，如果已经处理过，就不用再入队列，这就是去重。去重能大大优化复杂度。可以自己写哈希去重，缺点是很浪费空间。用 STL set 和 map 去重是更常见的做法。

DFS 常用的技巧是剪枝。如果某个搜索分支不符合要求，就剪去它不再深入搜索，这样能节省大量计算。

3.1.6 连通性判断

连通性问题是图论中的一个基本问题：寻找一张图中连通的点和边。很多图论题目或图论算法都需要判断连通性。判断连通性一般有 3 种方法：BFS、DFS、并查集。下面用一道例题介绍用 BFS 和 DFS 求解连通性问题。

例 3.1 全球变暖[①]

问题描述：有一张某海域 $N \times N$ 像素的照片，“.”表示海洋、“#”表示陆地，如下所示。

```
. . . . . .
. ##. . . .
. ##. . . .
. . . . ##.
. . ####.
. . . ###.
. . . . . .
```

其中，上、下、左、右 4 个方向上连在一起的一片陆地组成一座岛屿，如上面就有两座岛屿。

由于全球变暖导致海面上升，岛屿边缘一个像素的范围会被海水淹没。如果一块陆地像素与海洋相邻（上、下、左、右 4 个相邻像素中有海洋），它就会被淹没。例如，上述海域未来会变成

```
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . #. .
. . . . . .
. . . . . .
```

请计算：照片中有多少岛屿会被完全淹没？

① https://www.lanqiao.cn/problems/178/learning/

输入：第 1 行输入一个整数 $N(1\leqslant N\leqslant 1000)$；以下 N 行 N 列输入代表一张海域照片，照片保证第 1 行、第 1 列、第 N 行、第 N 列的像素都是海洋。

输出：输出一个整数表示答案。

这是基本的连通性问题。计算步骤：遍历一个连通块(找到这个连通块中所有的#，并标记已经搜过，不用再搜)；再遍历下一个连通块；直到遍历完所有连通块，统计有多少个连通块。

用暴力搜索解决连通性问题：逐个搜索连通块上的所有点，每个点只搜索一次。实现这种简单的暴力搜索，用 BFS 和 DFS 都可以，不仅很容易搜到所有点，而且每个点只搜索一次。

什么岛屿不会被完全淹没？若岛中有一块陆地(称为高地)，它周围都是陆地，那么这个岛屿不会被完全淹没。用 DFS 或 BFS 搜索出有多少个岛(连通块)，检查这个岛中有没有高地，统计那些没有高地的岛(连通块)的数量，就是答案。

计算复杂度：因为每个像素点只搜索一次且必须至少搜一次，共有 N^2 个点，复杂度为 $O(N^2)$，不可能更好了。

为了对比 DFS 和 BFS，先用 DFS 实现，再用 BFS 实现。

1. DFS 求解连通性问题

使用 DFS 搜索所有像素点，若遇到#，就继续搜索它周围的#。把搜索过的#标记为已经搜索过，不用再搜索。统计那些没有高地的岛的数量，就是答案。搜索时应该判断是不是出了边界。不过，题目已经说“照片保证第 1 行、第 1 列、第 N 行、第 N 列的像素都是海洋”那么就不用判断边界了，到了边界，发现是水，会停止搜索。

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1010;
char mp[N][N];                          //地图
int vis[N][N]={0};                      //标记是否搜索过
int d[4][2] = {{0,1}, {0,-1}, {1,0}, {-1,0}};       //4个方向
int flag;                               //用于标记这个岛是否被完全淹没
void dfs(int x, int y){
    vis[x][y] = 1;                      //标记这个#被搜索过,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++){           //继续搜索周围的陆地
        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[][]
                //继续搜索未搜索过的陆地,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++)          //搜索所有像素点
```

```
25          for(int j = 1; j <= n; j++)
26              if(mp[i][j] == '#' && vis[i][j] == 0){
27                  flag = 0;                   //假设这个岛被淹没
28                  dfs(i,j);                   //找这个岛中有没有高地,如果有,置 flag = 1
29                  if(flag == 0)  ans++;       //这个岛被淹没,统计被淹没岛的数量
30              }
31      cout << ans << endl;
32      return 0;
33  }
```

代码第 15 行中判断 vis[nx][ny]==0,如果没有这个判断条件,那么很多点会重复进行 DFS,导致超时。这是一种剪枝技术,叫作“记忆化搜索”。

2. BFS 求解连通性问题

BFS 的思路比较简单,代码如下。代码第 34 行中,看到一个 # 后,就用 BFS 扩展它周围的 #,所有和它相连的 # 属于一个岛。然后按前面 DFS 提到的方法找高地并判断是否淹没。BFS 的代码比 DFS 要复杂一点,因为用到了队列。这里直接用 STL queue,入队列的是坐标点 pair<int,int>。

```
 1  #include<bits/stdc++.h>
 2  using namespace std;
 3  const int N = 1010;
 4  char mp[N][N];
 5  int vis[N][N];
 6  int d[4][2] = {{0,1}, {0,-1}, {1,0}, {-1,0}};          //4 个方向
 7  int flag;
 8  void bfs(int x, int y) {
 9      queue<pair<int, int>> q;
10      q.push({x, y});
11      vis[x][y] = 1;                                      //标记这个 # 被搜索过
12      while (q.size()) {
13          pair<int, int> t = q.front();
14          q.pop();
15          int tx = t.first, ty = t.second;
16          if( mp[tx][ty+1] == '#' && mp[tx][ty-1] == '#' &&
17              mp[tx+1][ty] == '#' && mp[tx-1][ty] == '#'   )
18              flag = 1;                                   //上下左右都是陆地,不会淹没
19          for (int i = 0; i < 4; i++) {                   //扩展(tx,ty)的 4 个邻居
20              int nx = tx + d[i][0], ny = ty + d[i][1];
21              if(vis[nx][ny] == 0 && mp[nx][ny] == '#'){  //将陆地入队列
22                  vis[nx][ny] = 1;                        //注意:这一句必不可少
23                  q.push({nx, ny});
24              }
25          }
26      }
27  }
28  int main() {
29      int n;  cin >> n;
30      for (int i = 0; i < n; i++)     cin >> mp[i];
31      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;
}
```

【习题】[①]

(1) 力扣网站：

- DFS：https://leetcode-cn.com/tag/depth-first-search/
- BFS：https://leetcode-cn.com/tag/breadth-first-search/

(2) 洛谷网站：

- DFS：洛谷 P1219(八皇后[②])/P1019/P5194/P5440/P1378。
- BFS：P1162/P1443/P3956/P1032/P1126。

(3) poj 2488/3083/3009/1321/3278/1426/3126/3414/2251。

扫一扫

视频讲解

3.2 剪 枝

剪枝是搜索必用的优化手段，常常能把指数级的复杂度优化到近似多项式的复杂度。剪枝是一个比喻：把不会产生答案的或不必要的枝条“剪掉”。剪枝的关键在于剪枝的判断：什么枝该剪；在什么地方剪。

BFS 的剪枝通常使用判重，如果搜索到某一层时，出现重复的状态，就剪枝。例如，经典的八数码问题，核心就是去重，把曾经搜索过的八数码组合剪去。

DFS 的剪枝技术较多，有可行性剪枝、搜索顺序剪枝、最优性剪枝、排除等效冗余、记忆化搜索等等。

(1) 可行性剪枝：对当前状态进行检查，如果当前条件不合法就不再继续，直接返回。例如放最小的也放不下，或者放最大的也放不满。

(2) 搜索顺序剪枝：搜索树有多个层次和分支，不同的搜索顺序会产生不同的搜索树形态，复杂度也相差很大。

(3) 最优性剪枝：在最优化问题的搜索过程中，如果当前花费的代价已超过前面搜索到的最优解，那么本次搜索已经没有继续进行下去的意义，直接退出。

(4) 排除等效冗余：如果沿当前节点搜索它的不同分支，最后的结果是一样的，那么只

① 《算法竞赛入门到进阶》(清华大学出版社，罗勇军等著)第 4 章“搜索技术”讲解了一些经典题目：排列问题、子集生成和组合问题、八数码问题、八皇后问题、埃及分数等。

② 八皇后的解法很多。这里有一个有趣的解法，即用位运算求解(http://www.matrix67.com/blog/archives/266)。

搜索一个分支就够了。

(5) 记忆化搜索：在递归的过程中，有许多分支被反复计算，会大大降低算法的执行效率。用记忆化搜索，将已经计算出来的结果保存起来，以后需要用到时直接取出结果，避免重复运算，从而提高了算法的效率。记忆化搜索主要应用于动态规划，请参考本书“动态规划”这一章。

一道题目中可能用到多种剪枝技术，不过用不着刻意区分是哪种剪枝技术，总体思路就是减少搜索状态。虽然不是所有搜索题都需要剪枝，不过尽量考虑剪枝，概括为一句话：“搜索必剪枝，无剪枝不搜索”。

3.2.1 BFS 判重

BFS 剪枝的题目很多需要判重。BFS 的原理是逐步扩展下一层，把扩展出的下一层的点放入队列中处理。在处理上一层的同时，把下一层的点放到队列的尾部。在任意时刻，队列中只包含相邻两层的点。如果这些点都不同，只能把所有点放入队列。如果这些点有相同的，那么相同的点只处理一次就够了，其他相同的点不用重复处理，此时需要判重。例 3.2 是判重的例题。

例 3.2 跳蚱蜢[①]

问题描述：有 9 个盘子围成一个圆圈。其中 8 个盘子内装着 8 只蚱蜢，有一个是空盘。把这些蚱蜢顺时针编号为 1～8。每只蚱蜢都可以跳到相邻的空盘中，也可以再用点力，越过一个相邻的蚱蜢跳到空盘中。如果要使蚱蜢们的队形改为按照逆时针排列，并且保持空盘的位置不变(也就是 1 和 8 换位，2 和 7 换位，…)，至少要经过多少次跳跃？

这是一道八数码问题，八数码是经典的 BFS 问题。

本题用到了“化圆为线”的技巧。

直接让蚱蜢跳到空盘有点麻烦，因为有很多蚱蜢在跳。如果反过来看，让空盘跳，跳到蚱蜢的位置，就简单多了，因为只有一个空盘在跳。题目给的是一个圆圈，不好处理，可以“化圆为线”。把空盘标记为 0，那么有 9 个数字{0,1,2,3,4,5,6,7,8}，一个圆圈上的 9 个数字拉直成为一条线上的 9 个数字。这就是八数码问题，八数码有 9 个数字{0,1,2,3,4,5,6,7,8}，它有 9!=362880 种排列，不算多。

本题的初始状态是 012345678，终止状态是 087654321。从初始状态跳一次，下一个状态有 4 种情况，如图 3.2 所示。

① https://www.lanqiao.cn/problems/642/learning/

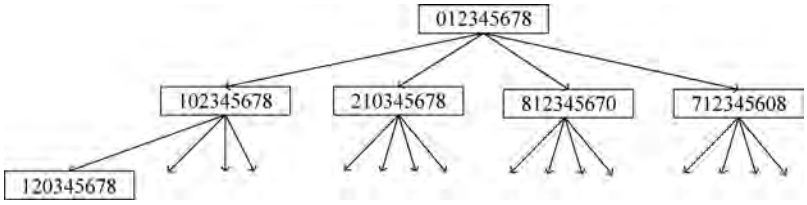


图 3.2 从起点开始的跳跃

用 BFS 扩展每层。每层就是蚱蜢跳了一次，扩展到某一层时发现终点 087654321，这一层的深度就是蚱蜢跳跃的次数。

如果写一个裸的 BFS，能运行出来吗？第 1 步到第 2 步有 4 种跳法；第 2 步到第 3 步有 4×4 种跳法；…；到第 20 步有 4^{20} 约 1 万亿种跳法。

必须判重，判断有没有重复跳，如果跳到一个曾经出现过的情况，就不用往下跳了。一共只有 9!= 362880 种情况。代码的复杂度是多少？在每层，能扩展出最少 4 种，最多 362880 种情况，最后算出的答案是 20 层，那么最多计算 20×362880＝7257600 次。在下面的 C++代码中统计实际的计算次数，是 1451452 次。

如何判重？用 STL 的 map 和 set 判重，效率都很好。另外，有一种数学方法叫作康托判重[①]，竞赛时一般不用。下面是例 3.2 的代码，其中有 map 和 set 两种判重方法。

```
#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
```

① 康托判重的详细讲解，请参考《算法竞赛入门到进阶》(罗勇军，郭卫斌著，清华大学出版社出版)4.3.2 节“八数码问题”。

```
                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 剪枝的应用

一道题目中可能用到多种剪枝技术，请通过以下例题掌握剪枝。例题的难度逐渐增加。

1. poj 3278

例 3.3 Catch that cow(poj 3278)

问题描述：在一条直线上，奶牛在 K 位置，农夫在 N 位置。农夫想抓到牛，他有 3 种移动方法：如他在 X 位置，他每次可以移动到 $X-1$、$X+1$、$2X$ 的位置。问农夫最少要移动多少次才能从 N 到达 K？

输入：输入两个整数 N 和 K。$0 \leqslant N, K \leqslant 100000$。

输出：最少移动次数。

这是从 N 到 K 的最短路径问题，显然用 BFS，每步有 3 个分支。本题使用**可行性剪枝**：如果农夫当前位置大于 K，那么农夫只能不断做 $X-1$ 操作，而不能使用变大的 $X+1$ 和 $2X$ 这两种操作。

2. 洛谷 P1118

例 3.4 数字三角形(洛谷 P1118)

问题描述:写出一个 1～n 的序列 a_i,然后每次将相邻两个数相加,构成新的序列,再对新序列进行这样的操作,显然每次构成的序列都比上一次的序列长度少 1,直到只剩下一个数字为止。下面是一个例子:

```
3  1  2  4
  4  3  6
    7  9
     16
```

现在倒着玩这个游戏,如果知道 n,知道最后得到的数字 sum,请求出最初序列 a_i,即为 1～n 的一个排列。若答案有多种可能,则输出字典序最小的那个。$n\leqslant 12$,sum$\leqslant$12345。

输入:输入两个正整数 n 和 sum。

输出:输出字典序最小的那个序列。

输入样例:	输出样例:
4 16	3 1 2 4

本题用暴力法会超时。对 1～n 这 n 个数做从小到大的全排列,对每个全排列进行三角形和的计算,判断是否等于 n。对每个排列进行三角形和计算复杂度为 $O(n^2)$。例如,第 1 行有 5 个数$\{a,b,c,d,e\}$,那么第 2 行计算 4 次,第 3 行计算 3 次…总次数为 $O(n^2)$。

$$\begin{array}{ccccccccc} a & & b & & c & & d & & e \\ & a+b & & b+c & & c+d & & d+e & \\ & & a+2b+c & & b+2c+d & & c+2d+e & & \\ & & & a+3b+3c+d & & b+3c+3d+e & & & \\ & & & & a+4b+6c+4d+e & & & & \end{array}$$

$n=12$ 时,共有 12! 约 4 亿种排列,总复杂度为 $O(n!\,n^2)$,显然会超时。

本题的解法是三角计算优化+剪枝。

(1) 三角计算优化。对排列进行三角形计算,并不需要按部就班地算,如$\{a,b,c,d,e\}$这 5 个数,直接计算最后一行的公式 $a+4b+6c+4d+e$ 就好了,复杂度为 $O(n)$。不同的 n 有不同的系数,例如 $n=5$ 的系数是$\{1,4,6,4,1\}$,提前算出所有 n 的系数备用。可以发现,这些系数正好是杨辉三角。

(2) 剪枝。即使有了杨辉三角的优化,总复杂度还是有 $O(n!\,n)$,所以必须进行**最优性剪枝**。对某个排列求三角形和时,如果前面几个元素和已经大于 sum,那么后面的元素就不用再算了。例如,$n=9$ 时,计算到排列$\{2,1,3,4,5,6,7,8,9\}$,如果前 5 个元素$\{2,1,3,4,5\}$求和已经大于 sum,那么后面的$\{6,7,8,9\}$～$\{9,8,7,6\}$都可以跳过,下一个排列从$\{2,1,3,4,6,5,7,8,9\}$开始。本题要求 sum$\leqslant$12345,和不大,用这个简单的剪枝方法可以通过。

(3) 可以用 DFS 求全排列,也可以直接用 STL 的 next_permutation()函数求全排列。

3. 洛谷 P1433

例 3.5　吃奶酪(洛谷 P1433)

问题描述：房间里有 n 块奶酪，给出每块奶酪的坐标。一只小老鼠要把它们都吃掉，它的初始坐标是(0,0)，问至少要跑多少距离？$1 \leqslant n \leqslant 15$。

输入：第 1 行输入一个整数，表示奶酪的数量 n；第 $2 \sim n+1$ 行中，每行输入两个实数，第 $i+1$ 行的实数表示第 i 块奶酪的坐标 (x_i, y_i)。

输出：输出一个实数，表示要跑的最短距离，保留两位小数。

本题是一个全排列问题，15 个奶酪有 15 个坐标，有 15! 共约 1.3 万亿种排列。如果用暴力法，用 DFS 搜索所有的排列，代码很容易写。在测试数据比较小的情况下，可以用**最优性剪枝**。在 DFS 中，用 sum 记录当前最短距离，每次计算新的距离时，如果大于 sum，就退出。剪枝的效率和测试数据有关，如果碰巧有很恶劣的数据，会超时。

提示　本题的标准解法是状态压缩 DP，它的复杂度为 $O(n2^n)$。

4. hdu 1010

例 3.6　Tempter of the bone(hdu 1010)

问题描述：一个迷宫有 $N \times M$ 格，有一些格子是地板，能走；有一些格子是障碍，不能走。给一个起点 S 和一个终点 D。一只小狗从 S 出发，每步走一块地板，在每块地板不能停留，而且走过的地板都不能再走。给定一个 T，问小狗能正好走 T 步到达 D 吗？

输入：有很多测试样例。每个测试中，第 1 行输入整数 $N, M, T (1<N, M<7, 0<T<50)$。后面 N 行中，每行输入 M 个字符，有这些字符可以输入：'X'：墙；'S'：起点；'D'：终点；'.'：地板。最后一行输入'0 0 0'，表示输入结束。

输出：每个测试，如果狗能到达，输出 YES，否则输出 NO。

用 BFS 还是 DFS？对于最短路径问题，应该用 BFS。但是如果要搜索所有路径，应该用 DFS，因为 DFS“一路到底”，天然就产生了一条路径，而 BFS 逐层推进，记录最短路径很方便，记录所有可能路径比较麻烦。

首先考虑暴力搜索的复杂度。在所有可能的路径中，看其中是否有长度为 T 的路径。直接搜索所有的路径，会超时。有多少可能的路径呢？本题图很小，但是路径数量惊人。$1<N, M<7$，最多有 36 个格子，设最长路径是 36，每个点有 3 个出口，那么就有 3^{36} 条路径，这是一个天文数字。即使在 DFS 加上限制条件，即格子不能重复走，也仍然会搜到百万条以上的路径。

本题最重要的技巧，网上称为“**奇偶剪枝**”。不过，本书认为“奇偶剪枝”这个说法不准

确，称为“**奇偶判断**”更合适，因为它并不需要在 DFS 内部剪枝，详见本节后面的讨论。

首先看两个容易发现的**可行性剪枝**，如下所示。

(1) 当前走了 k 步，如果 $k>T$，已经超过了限制步数，还没有到达 D 点，则剪掉。在 $k>T$ 的基础上，可以发现以下更好一点的剪枝。

(2) 设从起点 S 走了 k 步到达当前位置 (x,y)，而 (x,y) 到 D 点 (c,d) 的最短距离为 f，如果有 $k+f>T$，也就是 $T-k-f<0$，这说明剩下还允许走的步数比最短距离还少，肯定走不到了，剪掉。记 $\text{tmp}=T-k-f$。f 很容易求，它就是曼哈顿距离：$f=\text{abs}(c-x)+\text{abs}(d-y)$。这是理论上的最短距离，中间可能有障碍，不过不影响逻辑。

由于剪枝(2)比剪枝(1)严格，所以保留剪枝(2)就行了。

以上两种优化很有限，真正有用的是“奇偶剪枝”：若 tmp 为偶数，则可能有解；若 tmp 为奇数，肯定无解。

下面说明“奇偶剪枝”的原理。令 $\text{tmp}=T-k-f=T'-f$，T' 为当前位置 (x,y) 到 D 点要走的距离，那么 tmp 表示在最短路径之外还必须要走的步数，那么只能绕路了。比较简单的绕路方法是在最短路径上找两个相邻点 u 和 v，现在不直接走 $u\to v$，而是从 u 出发绕一圈再到 v，新路径是 $u\to\cdots\to v$；读者可以发现，在这种方格图上，原来的步数为 1，绕路后的步数一定比 1 大偶数步，也就是 tmp 是一个偶数。如果不用这个简单办法绕路，改用别的绕路方法，tmp 也是偶数。

其实，上述解释过于烦琐了，下面以图 3.3 解释更加透彻，而且还能得到更简洁的结果。实际上，在本题中，只需要对起点 S、终点 D 做一次奇偶判断就够了，DFS 内部不用再做。因为从 S 走到方格中的任意点 x，x 和 D 的奇偶性与 S 和 D 的奇偶性相同。所以，奇偶判断应该在 DFS 之前做，判断有解后再进行 DFS。DFS 内部的奇偶判断是多余的；奇偶判断并不能减少 DFS 内部搜索的次数，因为这是独立的两件事。

0	1	0	1	0	1
1	0	1	0	1	0
0	1	0	1	0	1
1	0	1	0	1	0
0	1	0	1	0	1

图 3.3 方格图的奇偶性

如图 3.3 所示，对每个方格用 0 和 1 进行交错标记。从 0 格子走一步只能到 1 格子，从 1 格子走一步只能到 0 格子。任取两个点为起点 S 和终点 D，如果它们都为 0 或 1，那么偶数步才能走到；如果它们一个为 0 一个为 1，那么奇数步才能走到。这就是奇偶判断的原理。

读者请判断，“奇偶剪枝”和“奇偶判断”哪种说法更准确？

所以，给定起点 S 和终点 D，以及限制的步数 T，可以立刻判断是否有解：①S 和 D 同为 0 或同为 1，T 为偶数，可能有解；T 为奇数，必定无解；②S 和 D 不同，T 为奇数，可能有解；T 为偶数，必定无解。

如果判断可能有解，有以下推论：从 S 出发到任意点 x，x 到 D 也可能有解，因为 x 和 D，与 S 和 D 的奇偶性相同。例如，S 和 D 都为 0，T 为偶数，可能有解；现在 S 走一步到 x，x 为 1，D 还是 0，T 也减 1 变为奇数，那么从 x 到 D 仍满足可能有解的判断。

但是，方格的 0/1 标记如何得到呢？或者换个说法：给定 S 和 D，如何判断它们是否相同呢？很简单，用曼哈顿距离就可以。如果 S 和 D 的曼哈顿距离 f 为奇数，说明 S 和 D 中

一个是 0 一个是 1；如果 f 为偶数，说明 S 和 D 同 0 或同 1。

在本题中，如果 $T-f$ 为奇数，则肯定无解。因为：假设 T 为奇数，那么 f 只能是偶数，也就是说，限制走奇数 T 步，但是 S 和 D 之间的路径是偶数步的，互相矛盾。

最后，通过以上分析可以知道，奇偶判断只能用在方格图上。方格中允许有不可走的障碍，这些障碍不影响逻辑正确性。

本题代码如下。

```
//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),剪枝(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()函数中做
    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
```

```
45          if(flag)  puts("YES");
46          else      puts("NO");
47      }
48      return 0;
49  }
```

5. 洛谷 P1120

例 3.7 小木棍(洛谷 P1120)

问题描述：乔治有一些同样长的小木棍，他把这些木棍随意砍成几段，直到每段的长都不超过 50。现在，他想把小木棍拼接成原来的样子，却忘记了自己开始时有多少根木棍和它们的长度。给出每段小木棍的长度，编程帮他找出原始木棍的最小可能长度。

输入：第 1 行输入一个整数 n，表示小木棍的个数；第 2 行输入 n 个整数，表示各木棍的长度 a_i。$1 \leqslant n \leqslant 65$，$1 \leqslant a_i \leqslant 50$。

输出：输出一个整数表示答案。

先考虑暴力法是否可行。尝试原始木棍所有可能的长度，看是否能拼接好这 N 个小木棍。例如，设原始木棍长度为 D，搜索所有木棍组合，如果能够把 N 个木棍都拼接成长度为 D 的木棍，则 D 就是一个合适的长度，在所有合适的长度中取最小值输出。用 DFS 搜索所有组合，对于一个 D 的检查，小木棍的组合复杂度为 $O(N!)$。

本题需要用到多种剪枝技术。

(1) **优化搜索顺序**。把小木棍按长度从大到小排序，然后按从大到小的顺序做拼接的尝试。对于给定的可能长度 D，从最长的小木棍开始拼接，在拼接时，继续从下一个较长的小木棍开始；持续这个操作，直到所有木棍都拼接成功或某个没有拼接成功为止。一旦不能拼接，这个 D 就不用再尝试。

(2) **排除等效冗余**。上面的优化搜索顺序中，是用贪心策略进行搜索。为什么这里可以用贪心策略？因为不同顺序的拼接是等效的，即先拼长的 x 再拼短的 y，与先拼短的 y 再拼长的 x 是一样的。

(3) 对长度 D 的优化。其实并不用检查大范围的 D，因为 D 是小木棍总长度的一个约数。例如，总长度为 10，那么 D 只可能是 1、2、5、10。计算小木棍的总长度，找到它的大于最长小木棍长度的所有约数，这就是原始木棍的可能长度 D。然后按从小到大排序，尝试拼接，如果成功，则输出结果，后面不再尝试。

6. hdu 2610

例 3.8 Sequence one(hdu 2610)

问题描述：给定一个序列，包含 n 个整数，每个整数不大于 2^{31}，输出它的前 p 个不递减序列，如果不够 p 个，就输出所有。

示例：有3个整数{1,3,2}，它的前5个不递减序列是{1}、{3}、{2}、{1,3}、{1,2}；输出时，首先按子序列长度排序，相同长度的按出现顺序排序，所以{3}在{2}前面，{1,3}在{1,2}前面。这个例子中没有长度为3的不递减序列。

输入：包括多个测试，每个测试输入两个整数 n 和 p。$1<n\leqslant 1000$，$1<p\leqslant 10000$。

输出：对于每个测试，输出答案。

下面用两种方法求解。

1）暴力法

暴力法求解分为3部分：生成所有子序列、去重、去掉递减序列。

（1）生成所有子序列。用DFS编码比较简单，题目求不同长度的序列，可以按长度分别DFS。

```
void dfs(int len, int pos){
…
for(int i = pos;i < n;i++)              // pos 表示当前位置,从 pos 位置开始找子序列
     dfs(len,i);
…
}
int main(){
…
    for(int len = 1;len < = n;len++)    //len 表示子序列的长度,每次搜索一种长度的子序列
    dfs(len,0);
…
}
```

（2）去重。简单的办法是用STL set，理论上对一个子序列进行判重的复杂度为 $O(\log_2 n)$。

（3）去掉递减序列。在做DFS时，如果子序列中的下一个元素比上一个元素大，就退出。

上述步骤看起来不错，但是会超时。虽然只需要输出前 p 个子序列，但是要搜索的范围远远超过 p。去重很花时间，去掉递减序列也很花时间，长度为2及以上的子序列有大量是递减的。

2）去重的优化和可行性剪枝

本题的去重和去掉递减序列都可以优化。

（1）去重的优化。

思路：用某元素 a 为首元素，在原始序列中生成不递减子序列后，后面如果再遇到相等的 a，就不用再生成子序列了，因为前面已经用 a 在整个范围内搜索过了；这个思路可以推广到第2个元素、第3个元素，等等。

下面以序列 $A[]=\{1,2,1,5,1,4,1,7\}$ 为例说明，它的不递减子序列有{1}、{2}、{5}、{4}、{7}、{1,2}、{1,1}、{1,5}、{1,4}、{1,7}、{2,5}、{2,4}、{2,7}、{5,7}，等等。

① 以 $A[0]=1$ 为首，生成了{1}、{1,2}、{1,1}、{1,5}、{1,4}等序列；下次准备以 $A[2]=1$ 为首生成子序列时，发现前面有 $A[0]=A[2]=1$，那么就丢弃以 $A[2]=1$ 为首的

所有子序列，因为前面已经用 $A[0]=1$ 为首，在整个序列中得到了{1}、{1,5}、{1,4}等序列。

② 以 $A[3]=5$ 为首的子序列，确定子序列的第 2 个元素时，在 $A[3]$后面的{1,4,1,7}范围内，按方法①操作，如检查{1,7}的 1 时，这个 1 已经在{1,4,1,7}的第 1 个位置出现过，所以应该丢弃。

复杂度分析：上面的去重方法，对一个子序列做一次判重的复杂度为 $O(n)$，似乎比 STL set 的 $O(\log_2 n)$差；不过，前者剪去了很多子序列，需要判重的子序列比后者少很多。

(2) 去掉递减序列的剪枝。

如果短的子序列没有合法的，那么更长的也不合法。例如，搜索长度为 4 的子序列，发现没有非递减的，那么大于 4 的非递减子序列也不存在，剪去。

hdu 2611 是类似的题目，请读者自己了解。

例 3.9 Sequence two(hdu 2611)

问题描述：给定一个序列，包含 n 个整数，每个整数不大于 2^{31}，按字典序输出它的前 p 个不递减序列，如果不够 p 个，就输出所有。

不递减序列的例子：有 3 个整数{1,3,2}，它的所有 5 个序列是{1}、{2}、{3}、{1,2}、{1,3}，按字典序输出；注意{1,2,3}不是它的子序列，因为不能改变元素的顺序。$1<n\leqslant 100$，$1<p\leqslant 100000$。

7. poj 2676

例 3.10 Sudoku(poj 2676)

问题描述：九宫格问题，又称为数独问题。把一个 9 行 9 列的网格再细分为 9 个 3×3 的子网格，要求每行、每列、每个子网格内都只能填 1～9 中的一个数字，每行、每列、每个子网格内都不允许出现相同的数字。给出一个填写了部分格子的九宫格，要求填完九宫格并输出，如果有多种结果，则只需输出其中一种。

本题的总体思路是用 DFS 搜索每个空格子。写代码时用到一个小技巧：用位运算记录格子状态。每行、每列、每个子网格，分别用一个 9 位的二进制数保存哪些数字还可以填。对于每个位置，把它在的行、列，九宫格对应的数取 & 运算就可以得到剩余哪些数可以填，并用 lowbit()函数取出能填的数。在这些操作的基础上，用到以下两种剪枝技术。

(1) **优化搜索顺序剪枝**。从最容易确定数字的行(或列)开始填数，也就是 0 最少的行(或列)；在后续每个状态下，也选择 0 最少的行(或列)填数。

(2) **可行性剪枝**。每格填的数只能是对应行、列和子网格中没出现过的。

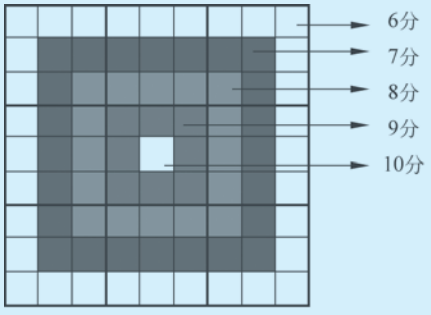
洛谷 P1074 是本题的扩展。

8. 洛谷 P1074

例 3.11　靶形数独(洛谷 P1074)

问题描述：靶形数独的方格与普通数独一样，在 9×9 的大九宫格中有 9 个 3×3 的小九宫格(用粗黑色线隔开的)。在这个大九宫格中，有些数字是已知的，根据这些数字，利用逻辑推理，在其他空格中填入数字 1～9。每个数字在每个小九宫格内不能重复出现，每个数字在每行、每列也不能重复出现。但靶形数独有一点与普通数独不同，即每个方格都有一个分值，而且如同一个靶子一样，离中心越近则分值越高。

比赛要求：每个人必须完成一个给定的数独(每个给定数独可能有不同的填法)，而且要争取更高的总分数。而这个总分数即每个方格上的分值和完成这个数独时填在相应格上的数字的乘积的总和。如图 3.4 所示，在这个已经填完数字的靶形数独游戏中，总分数为 2829。游戏规定以总分数的高低决出胜负。



7	5	4	9	3	8	2	6	1
1	2	8	6	4	5	9	3	7
6	3	9	2	1	7	4	8	5
8	6	5	4	7	9	1	7	3
9	7	2	3	5	1	6	4	8
4	1	3	8	7	6	5	2	9
5	4	7	1	8	2	3	9	6
2	9	1	7	6	3	8	5	4
3	8	6	5	9	4	7	1	2

图 3.4　靶形数独

此题较难，几个关键点如下。

(1) 九宫格的表示。把每个格子对应的分数以及每个格子属于哪个小九宫格用二维数组打表，方便搜索时使用。

(2) **优化搜索顺序剪枝**。从最容易确定数字的行(或列)开始填数，也就是 0 最少的行(或列)；在后续每个状态下，也选择 0 最少的行(或列)填数。

(3) **可行性剪枝**。每格填的数只能是对应行、列和子网格中没出现过的。

【习题】

(1) 洛谷 P1120/P1312/P1074。

(2) poj 1010/2362/1011/1416/2676/1129/1020/3411/1724。

扫一扫

视频讲解

3.3　洪水填充

洪水填充[①]算法是搜索的一个简单应用。一张图上有多个区域，不同的区域用不同颜

① 英文是 Flood Fill 或 Seed Fill。洪水填充(Flood Fill)和边界填充(Boundary Fill)是解决区域填充(Area Filling)问题的算法。

色区分，同一个区域的所有点的颜色(oldColor)都是相同的。给定图上的一个点，称为种子点，然后从种子点出发，把种子点所属的封闭区域用新颜色(fillColor)填充，这就是“洪水填充”。

拍电影时常用绿幕做背景，成片时再把绿幕抠掉，这时可以用洪水填充算法。不过洪水填充算法比较低效，真正的抠图需要用高效的算法。

如图 3.5 所示，左图方框内有 3 个区域：心形的边界、心形内部、心形外部。心形内部和外部的颜色都是白色，但是被黑色的心形边界隔开，所以不在一个区域中。若种子点在心形内部，用灰色填充后的效果见右图。

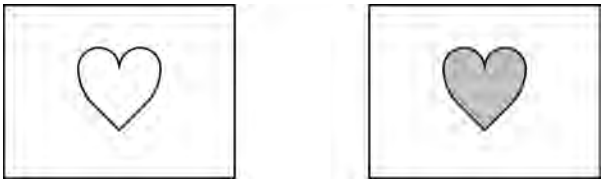


图 3.5 洪水填充

填充过程像洪水一样从种子开始向四周蔓延，首先扩散到它的邻居，然后再扩散到邻居的邻居，这实际上是一个寻找连通块的过程。根据图形的定义，邻居有 4 个(上、下、左、右共 4 个)或 8 个(加上对角线方向的 4 个)。

洪水填充的编程用 BFS 和 DFS 都可以。洪水扩散过程符合 BFS 的原理，不过用 DFS 编码更简单，以下是 4 邻居图的 DFS 代码。

```
void floodfill(int x, int y, int fillColor, int oldColor){ //从种子点的坐标开始
    if(check(x,y) == True && color(x,y) == old_color){//check(x,y)函数检查是否越过图的边界
        setColor(x, y, fillColor);                    //给这个点涂色
        floodfill(x+1,y, fillColor, old_color);       //递归 4 个邻居
        floodfill(x-1,y, fillColor, old_color);
        floodfill(x,y+1, fillColor,old_color);
        floodfill(x,y-1, fillColor,old_color);
    }
}
```

设图是一个 $n \times n$ 的矩阵，洪水填充算法的时间、空间复杂度都为 $O(n^2)$。若 n 较大，用 DFS 可能导致栈溢出，此时可以改用 BFS。BFS 一圈一圈向外扩散，最大的外圈有 $4n$ 个点，编码所使用的队列长度最大为 $4n$。

下面给出 3 道例题。

1. hdu 1312

例 3.12 Red and black(hdu 1312)

问题描述：有一个长方形的房间，铺着方形瓷砖，每块瓷砖都是红色或黑色。一个人站在黑色的瓷砖上，他可以按上、下、左、右方向移动到相邻的瓷砖。但他不能在红砖上移动，他只能在黑砖上移动。编程计算他可以达到的黑色瓷砖的数量。

输入：第 1 行输入两个正整数 W 和 H，分别表示 x 方向和 y 方向上的瓷砖数量，W 和 H 均不超过 20。下面输入 H 行，每行包含 W 个字符，每个字符表示一块瓷砖的颜色。字符表示：'·'表示黑色瓷砖；'#'表示红色瓷砖；'@'代表黑瓷砖上的人，在数据集中只出现一次。

输出：输出一个数字，表示从初始瓷砖能到达的瓷砖总数量（包括起点）。

题解 这是一道简单的洪水填充题。从中心点'@'出发，用 BFS 或 DFS 搜索与它连通的'·'即可[①]。

2. poj 2227

例 3.13 The wedding juicer（poj 2227）

问题描述：在地面上用砖块修占地面积为 $W\times H$ 的建筑，$3\leqslant W\leqslant 300$，$3\leqslant H\leqslant 300$，每 1×1 的单位地面上有一块砖，第 i 块砖的高度为 B_i，$1\leqslant B\leqslant 1000000000$，砖与砖之间紧密贴合。计算这块地面能容纳多大容量的水。

题解 把这块地面看作一个不规则的大水桶，其中有很多局部能储水，每个能储水的局部，其水面不能高于这个局部的边界上最矮的砖块。最简单的办法是逐个检查砖块，统计它周边的局部储水情况。但是这样比较复杂，为了简化逻辑，采用“从四周堵中央”的办法。

从边界（$W\times H$ 的最外面一圈）开始，找到边界上的最矮砖块 j，检查它的邻居砖块 k：①如果 $B_j\leqslant B_k$，k 上不能储水，将 k 标记为新的边界；②如果 $B_j>B_k$，该砖块可以储水，储水容量增加 B_j-B_k，把 k 的高度改为 j 的高度，然后也把 k 标记为新的边界。删除 j，后续不再处理。每次检查新边界，处理其中最矮的；逐步缩小边界，直到所有砖块被检查过。

编码时用优先队列处理边界，队列中的砖块始终表示当前的边界，队首是最矮的砖块。先把边界上所有的砖块放进优先队列，然后开始处理队列。从队首的最矮砖块 j 开始，把 j 弹出队列然后检查它的邻居砖块：比 j 高的标记为新边界并加入队列；比 j 矮的，统计储水量，修改高度，标记为新边界，加入队列。队列为空时计算结束。

3. 洛谷 P1514

例 3.14 引水入城（洛谷 P1514）

问题描述：在一个遥远的国度，一侧是风景秀美的湖泊，另一侧则是漫无边际的沙漠。该国的行政区划十分特殊，刚好构成一个 $N\times M$ 的矩形，其中每个格子都代表一座城市，第 1 行在湖边，第 N 行在沙漠边上，每座城市都有一个海拔高度。

为了使尽量多的居民有水，需要在城市建造水利设施。水利设施有两种，分别为蓄水

① 《算法竞赛入门到进阶》（罗勇军，郭卫斌著，清华大学出版社出版）第 44 页用 BFS、第 53 页用 DFS 解析了此题。

厂和输水站。蓄水厂的功能是利用水泵将湖泊中的水抽取到所在城市的蓄水池中。只有与湖泊毗邻的第 1 行的城市可以建造蓄水厂。输水站的功能则是通过输水管线利用高度落差，将湖水从高处向低处输送。因此，一座城市能建造输水站的前提是存在比它海拔更高且拥有公共边的相邻城市，已经建有水利设施。第 N 行的城市靠近沙漠，是该国的干旱区，所以要求其中的每座城市都建有水利设施。这个要求能否满足呢？如果能，请计算最少建造几个蓄水厂；如果不能，求干旱区中不可能建有水利设施的城市数目。

第 1 个问题“是否能满足最后一排都有水”是简单的洪水填充，用 BFS 或 DFS 检查第 1 排的每个点，把这些点看作种子点，扩散到比它们海拔低的邻居，直到最后一排，检查最后一排是否全部被覆盖到即可。

第 2 个问题是最少建几个蓄水厂。本题的特殊性在于，第 1 排的每个点所能覆盖的最后一排的点是有先后关系的。例如，第 1 排左边第 1 点覆盖了最后一排的$[L,R]$点，那么第 1 排第 2 点能覆盖的最后一排的点肯定大于或等于$[L,R]$。这一点容易理解，若第 1 点和第 2 点的水流都能流到最后一排，第 1 点的水不会比第 2 点的水流得更远。

本题的编码有点麻烦，请仔细练习。

【习题】

(1) hdu 5319/4574/1240/6113。

(2) 洛谷 P1506/P1162/P1649。

(3) poj 1979/3026/2157。

扫一扫

视频讲解

3.4 BFS 与最短路径

最短路径问题是最著名的图论问题，有很多不同的场景和算法。在一种特殊的场景中，BFS 也是极为优秀的最短路径算法，这种场景就是所有的相邻两个点的距离相等，一般把这个距离看作 1。此时，BFS 是最优的最短路径算法，查找一次从起点到终点的最短距离的计算复杂度为 $O(m)$，m 为图上边的数量，因为需要对每条边做一次检查。关于空间复杂度，用邻接表存储图的复杂度为 $O(n+m)$，另外需要使用一个 $O(n)$的队列，n 为点的数量。

提示 如果两点之间距离不相等，就不能用 BFS 了，需要用 Dijkstra 等通用算法。

BFS 的特点是逐层扩散，也就是按最短路径扩散出去。向 BFS 的队列中加入邻居节点时，是按距离起点远近的顺序加入的：先加入距离起点为 1 的邻居节点，再加入距离为 2 的邻居节点，依此类推。搜索完一层，才会继续搜索下一层。一条路径是从起点开始，沿着每层逐步向外走，每多一层，路径长度就加 1。那么，所有长度相同的最短路径都是从相同的层次扩散出去的。

求最短路径时，常见的问题有两个：①最短路径有多长？答案显然是唯一的；②最短路径经过了哪些点？由于最短路径可能不只一条，所以题目往往不要求输出，如果要求输

出，一般是要求输出字典序最小的那条路径。

下面用一道例题介绍最短路径的计算和最短路径的打印。

例 3.15　迷宫[①]

问题描述：给出一个迷宫的平面图，其中标记为 1 的为障碍，标记为 0 的为可以通行的地方，如下所示。

010000
000100
001001
110000

迷宫的入口为左上角，出口为右下角，在迷宫中，只能从一个位置走到这个它的上(U)、下(D)、左(L)、右(R)4 个方向之一。对于上面的迷宫，从入口开始，可以按 DRRURRDDDR 的顺序通过迷宫，一共 10 步。对于一个更复杂的迷宫(30 行 50 列)，请找出一种通过迷宫的方式，其使用的步数最少。在步数最少的前提下，请找出字典序最小的一个作为答案。

注意：在字典序中 D<L<R<U。

本题是基本的 BFS 搜索最短路径。BFS 是最优的算法，每个点只搜索一次，即入队列和出队列各一次。题目要求返回字典序最小的最短路径，那么只要在每次扩散下一层(向 BFS 的队列中加入下一层的节点)时，都按字典序 D<L<R<U 的顺序加下一层的节点，那么第 1 个搜索到的最短路径就是字典序最小的。

本题的关键是路径打印，下面给出两种打印方法。

(1) **简单方法**，适合小图。每扩展到一个点 v，都在 v 上存储从起点 s 到 v 的完整路径 path。到达终点 t 时，就得到了从起点 s 到 t 的完整路径。在下面的代码中，在每个节点上记录从起点到这个点的路径。到达终点后，用 cout << now.path << endl 就打印出了完整路径。这样做的缺点是会占用大量空间，因为每个点上都存储了完整的路径。

(2) **标准方法**，适合大图。其实不用在每个节点上存储完整路径，而是在每个点上记录它的前驱节点就够了，这样从终点能一步步回溯到起点，得到一条完整路径。这种路径记录方法称为“标准方法”。注意看代码中的 print_path()，它是递归函数，先递归再打印。从终点开始，回溯到起点后，再按从起点到终点的顺序，正序打印出完整路径。

下面的代码中包含了两种方法，用“(1)简单方法”和“(2)标准方法”区分。

```
#include<bits/stdc++.h>
using namespace std;
struct node{
    int x;
    int y;
//(1)简单方法
```

① https://www.lanqiao.cn/problems/602/learning/

```
    string path;                //path 记录从起点(0,0)到点(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 表示已经搜索过,不用再搜索

//(2)标准方法
char pre[31][51];               //用于查找前驱点,如 pre[x][y] = 'D'表示上一个点
                                //向下走一步到了(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,向上 U
    if(pre[x][y] == 'L')  print_path(x,  y+1);  //回溯,向右 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;                              //标记起点被搜索过
    queue<node> q;
    q.push(start);                              //把第 1 个点放入队列,开始 BFS
    while(!q.empty()){
        node now = q.front();                   //取出队首
        q.pop();
        if(now.x == 29 && now.y == 49){ //第 1 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++){                 //扩散邻居节点
            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 表示已经搜索过;  mp = 1 表示是障碍
            vis[next.x][next.y] = 1;            //标记被搜索过
//(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(){
```

```
59      for(int i = 0;i < 30;i++)   cin >> mp[i];              //读题目给的地图数据
60      bfs();
61  }
```

如果图上的每两个邻居节点之间的长度不同，上述普通的 BFS 做法就行不通了。这种一般性的最短路径算法，需要结合优先队列，具体做法详见 3.6 节。3.6 节实际上讲解图论的 Dijkstra 算法。

扫一扫

视频讲解

3.5 双向广搜

双向广搜的原理很简单：把从起点 s 到终点 t 的单向搜索改为分别从 s 出发的正向搜索和从 t 出发的逆向搜索。使用双向广搜时需要做两个判断：①能不能使用双向广搜；②双向广搜是否能显著改善算法复杂度。

3.5.1 双向广搜的原理和复杂度分析

1. 原理

双向广搜的应用场合：有确定的起点和终点，并且能把从起点到终点的单个搜索，变换为分别从起点出发和从终点出发的“相遇”问题，此时可以用双向广搜。从起点 s（正向搜索）和终点 t（逆向搜索）同时开始搜索，当两个搜索产生相同的一个子状态 v 时就结束。得到的 s-v-t 是一条最佳路径，当然，最佳路径可能不止这一条。

和普通 BFS 一样，双向广搜在搜索时并没有“方向感”，所谓“正向搜索”和“逆向搜索”其实仍然是盲目的，它们分别从 s 和 t 逐层扩散出去，直到相遇为止。

2. 复杂度分析

与只做一次 BFS 相比，双向广搜能在多大程度上改善算法复杂度呢？下面以网格图和树形结构为例，推出一般性结论。

1）网格图

用 BFS 求图 3.6 中 s 和 t 之间的最短路。图 3.6(b)是双向广搜，在中间的五角星位置相遇。

设两点的距离为 k。图 3.6(a)的 BFS，从起点 s 扩展到 t，一共访问了 $2k(k+1)\approx 2k^2$ 个节点；图 3.6(b)的双向 BFS，相遇时一共访问了约 k^2 个节点。两者差一倍，改善**并不明显**。

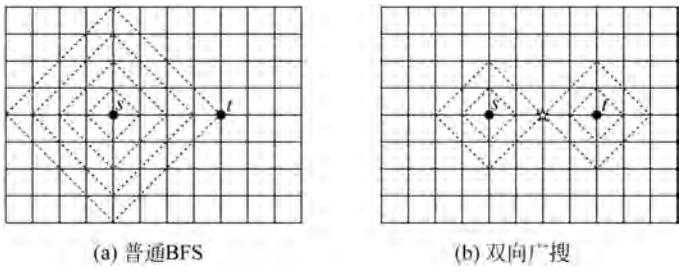
在这个网格图中，BFS 从第 k 扩展到第 $k+1$ 层，节点数量是**线性增长**的。

2）树形结构

以二叉树为例，如图 3.7 所示，求根节点 s 到最后一行的黑点 t 的最短路。

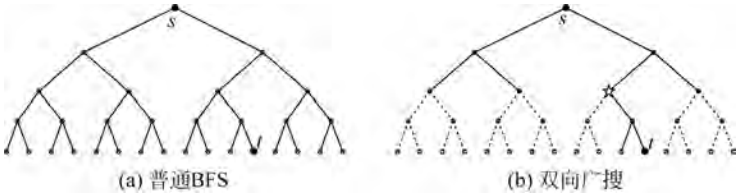
普通 BFS 从第 1 层到第 $k-1$ 层，共访问 $1+2+\cdots+2^{k-1}\approx 2^k$ 个节点。双向 BFS 分

别从上向下和从下向上进行 BFS，在五角星位置相遇，共访问约 $2\times 2^{k/2}$ 个节点。双向广搜相比做一次 BFS 优势巨大。



(a) 普通BFS　　(b) 双向广搜

图 3.6　网格图搜索



(a) 普通BFS　　(b) 双向广搜

图 3.7　二叉树搜索

在二叉树的例子中，BFS 扩展的第 k 和第 $k+1$ 层，节点数量是**指数增长**的。

从上面两个例子得出以下一般性结论。

(1) 做 BFS 扩展时，下一层节点(一个节点表示一个状态)数量增加越快，双向广搜越有效率。

(2) 是否用双向广搜代替普通 BFS，除了节点增长数以外，还应根据总状态数量的规模来决定。双向广搜的优势，从根本上说，是能减少需要搜索的状态数量。有时虽然下一层数量是指数增长的，但是由于**去重**或限制条件，总状态数并不多，也就没有必要使用双向广搜。例如 3.5.3 节的例题 hdu 1195，密码范围为 1111～9999，共约 9000 种，用 BFS 搜索时，最多有 9000 个状态进入队列，就没有必要使用双向广搜；而例题 hdu 1401，可能的棋局状态有 1500 万种，走 8 步棋会扩展出 16^8 种状态，相当于扩展到所有可能的棋局，此时应该使用双向广搜。

很多教材和网文讲解双向广搜时，常用八数码问题做例子。八数码共有 $9!=362880$ 种状态，不太多，用普通 BFS 也可以，3.2 节已经讲过八数码问题。不过，用双向广搜更好，因为八数码每次扩展，下一层的状态数量是上一层的 2～4 倍，比二叉树的增长还快，效率的提升也就更明显。

3.5.2　双向广搜的两种实现

双向广搜使用的队列有两种实现方法。

(1) 合用一个队列。正向 BFS 和逆向 BFS 用同一个队列，适合正、反两个 BFS 平衡的情况。正向搜索和逆向搜索交替进行，两个方向的搜索交替扩展子状态，先后入队，直到两个方向的搜索产生相同的子状态，即相遇了，结束。这种方法适合正、反方向扩展的新节点

数量差不多的情况，如八数码问题。

（2）分成两个队列。正向 BFS 和逆向 BFS 的队列分开，适合正、反两个 BFS 不平衡的情况。让子状态少的 BFS 先扩展下一层，另一个子状态多的 BFS 后扩展，可以减少搜索的总状态数，尽快相遇。

正向搜索和逆向搜索是否一定能够相遇？什么时候停止搜索？讨论以下两种情况。

（1）如果从起点 s 到终点 t 之间存在一条路径，那么从 s 开始的正向搜索和从 t 开始的逆向搜索，一定会在某点相遇（队列为空之前相遇），此时以相遇为终止条件。

（2）如果不存在从 s 到 t 的路径，那么肯定不能相遇，此时只要一个方向停止了搜索（这个方向的队列为空），就可以停止了，另一个方向继续搜索是做无用功。

综合起来，双向广搜的基本逻辑是在一个队列为空之前，若相遇，则说明有解，停止搜索并返回答案；若一个队列为空时还未相遇，说明无解，停止搜索。

和普通 BFS 一样，双向广搜在扩展队列时也需要处理去重问题。把状态入队列时，先判断这个状态是否曾经入队，如果重复了，就丢弃。

3.5.3 双向广搜例题

下面是几道双向广搜的经典题目。

1. hdu 1195

例 3.16 Open the lock（hdu 1195）

问题描述：打开密码锁。密码由 4 位数字组成，数字为 1～9。可以在任何数字上加上 1 或减去 1，当'9'加 1 时，数字变为'1'；当'1'减 1 时，数字变为'9'。相邻的数字可以交换。每个动作是一步。任务是使用最少的步骤打开锁。注意：最左边的数字不是最右边的数字的邻居。

输入：第 1 行输入整数 T，表示测试用例个数。每个测试包括两行，每行输入：四位数 N，表示密码锁初始状态；四位数 M，表示开锁的密码。

输出：对于每个测试用例，打印一个整数，表示最少的步骤。

题目中的 4 位数字，走一步能扩展出 11 种情况；如果需要走 10 步，就可能有 11^{10} 种情况，数量非常多，看起来用双向广搜能大大提高搜索效率。不过，本题用普通 BFS 也可以，因为并没有 11^{10} 种情况，密码范围为 1111～9999，只有约 9000 种。用 BFS 搜索时，最多有 9000 个状态进入队列，没有必要使用双向广搜。密码进入队列时，应去重，去掉重复的密码。读者可以用这一题练习双向广搜。

2. hdu 1401

这是经典的双向广搜例题。

例 3.17 Solitaire(hdu 1401)

问题描述：在8×8的方格中放4颗棋子在初始位置，给定4个最终位置，问在8步内是否能从初始位置走到最终位置。规则：每个棋子能上、下、左、右移动，若4个方向已经有一棋子，则可以跳到下一个空白位置。例如，图3.8中(4,4)位置的棋子有4种移动方法。

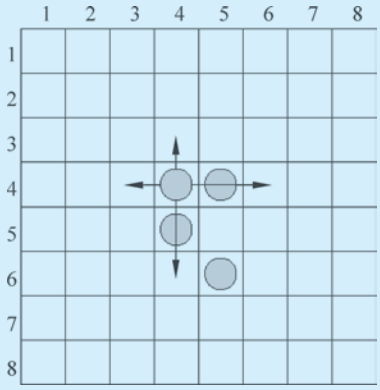


图3.8 棋子移动方法

在8×8的方格中放4颗棋子，有64×63×62×61≈1500万种棋局。走一步棋，4颗棋子共有16种走法，连续走8步棋，会扩展出16^8种棋局，16^8>1500万，走8步可能会遍历到1500万棋局。

此题应该使用双向广搜。从起点棋局走4步，从终点棋局走4步，如果能相遇就有一个解，共扩展出$2\times16^4=131072$种棋局，远远小于1500万。

本题也需要处理去重问题，扩展下一个新棋局时，看它是否在队列中处理过。用哈希的方法，定义char vis[8][8][8][8][8][8][8][8]表示棋局，其中包含4颗棋子的坐标。vis=1表示正向搜索过这个棋局，vis=2表示逆向搜索过。例如，4个棋子的坐标是(a.x,a.y)、(b.x,b.y)、(c.x,c.y)、(d.x,d.y)，那么vis[a.x][a.y][b.x][b.y][c.x][c.y][d.x][d.y]=1表示这个棋局被正向搜索过。

4颗棋子需要先排序，然后再用vis记录。如果不排序，一个棋局就会有很多种表示，不方便判重。

char vis[8][8][8][8][8][8][8][8]用了8^8B=16MB空间。不能定义为int型，占用64MB空间，超过题目的限制。

3. hdu 3095

例 3.18 Eleven puzzle(hdu 3095)

问题描述：有13个格子的拼图，数字格可以移动到黑色格子，如图3.9所示。左图是开始局面，右图是终点局面。一次移动一个数字格，问最少移动几次可以完成。

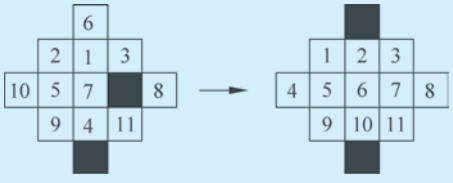


图3.9 数字拼图

可能的局面有 13！种，数量极大。只用一个 BFS，复杂度过高。每次移动一个黑格，移动方法最少一种，最多 8 种。如果移动 10 次，那么最多有 $8^{10}\approx10$ 亿种局面。用双向广搜能减少到 $2\times8^5=65536$ 种局面。判重可以用哈希，或者用 STL map。

4. 洛谷 P1032

例 3.19　字串变换（洛谷 P1032）

问题描述：已知有两个字串 A 和 B，以及一组字串变换的规则（至多 6 个规则）：

$A_1\rightarrow B_1$

$A_2\rightarrow B_2$

规则的含义为：在 A 中的子串 A_1 可以变换为 B_1，A_2 可以变换为 B_2，…

例如，A=abcd，B=xyz，变换规则为

abc→xu，ud→y，y→yz

则此时，A 可以经过一系列的变换变为 B，其变换过程为

abcd→xud→xy→xyz

共进行了 3 次变换，使 A 变换为 B。

给定字符串 A 和 B 以及变换规则，问能否在 10 步内将 A 变换为 B，并输出最少的变换步数。字符串长度的上限为 20。

本题若用普通 BFS 进行遍历，BFS 的每层扩展 6 个规则，经过 10 步，共有 $6^{10}\approx6000$ 万次变换。如果改用双向广搜，可以用 $2\times6^5=15552$ 次变换搜索完 10 步。

双向广搜的编码，用两个队列分别处理正向 BFS 和逆向 BFS。由于起点和终点的字符串不同，它们扩展的下一层数量也不同，也就是进入两个队列的字符串的数量不同，先处理较小的队列，可以加快搜索速度。代码示例如下。

```
//完整代码参考 https://blog.csdn.net/qq_45772483/article/details/104504951
void bfs(string A, string B){          //起点是 A,终点是 B
    queue < string > qa, qb;           //定义两个队列
    qa.push(A);                        //正向队列
    qb.push(B);                        //逆向队列
    while(qa.size() && qb.size()){
        if (qa.size() < qb.size())     //如果正向 BFS 队列小,先扩展它
            extend(qa, ...);           //扩展时,判断是否相遇
        else                           //否则扩展逆向 BFS
            extend(qb, ...);           //扩展时,判断是否相遇
    }
}
```

5. poj 3131

立体八数码（Cubic Eight-Puzzle）问题，状态多，代码长，是一道难题。

【习题】

洛谷 P3067/P4799/P5195。

扫一扫

视频讲解

3.6 BFS 与优先队列

BFS 的代码实现需要用到队列，在不同场景中使用普通队列或优先队列。本节介绍使用优先队列的经典 BFS 算法，即一般性的最短路径算法，这种算法实际上是图论的 Dijkstra 算法。

1. 优先队列

本书第 1 章介绍了普通队列和优先队列。普通队列中的元素是按先后顺序进出队列的，先进先出。在优先队列中，元素被赋予了优先级，每次弹出队列的是具有最高优先级的元素。优先级根据需求来定义，如定义最小值为最高优先级。

优先队列有多种实现方法。最简单的是暴力法，在 n 个数中扫描最小值，复杂度为 $O(n)$。暴力法不能体现优先队列的优势，优先队列一般用堆实现，插入元素和弹出最高优先级元素，复杂度都为 $O(\log_2 n)$。虽然基于堆的优先队列很容易手写，不过竞赛中一般不用自己写，而是直接用 STL 的 priority_queue。

2. 用 BFS 结合优先队列求解一般性最短路径问题

在 3.4 节中，介绍了边权为 1 的最短路径的求解。对于边权不等于 1 的普通图的最短路径问题，可以用 BFS 结合优先队列求解。

下面介绍“BFS＋优先队列”求最短距离的算法步骤。以图 3.10 为例，起点是 A，求 A 到其他节点的最短路径。图的节点总数为 n，边的总数为 m。图中边上的数字是边权，边权的实例有长度、费用等。一条路径的总权值等于路径上所有边权的和。

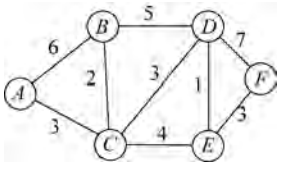


图 3.10 网络图

基于“BFS＋优先队列”的算法用到了贪心策略。从起点 A 开始，逐层扩展它的邻居，放到优先队列中，并从优先队列中弹出距离 A 最近的点，就得到了这个点到 A 的最短距离；当新的点放入队列中时，如果经过这个点，使队列中它的邻居到 A 更近，就更新这些邻居点到 A 的距离。

以图 3.10 为例，步骤如下。

(1) 开始时，把起点 A 放入优先队列 Q 中：$\{A_0\}$。下标表示从 A 出发到这个点的路径长度，A 到自己的距离为 0。

(2) 从队列中弹出最小值，即 A，然后扩展 A 的邻居节点，放入优先队列 Q 中：$\{B_6, C_3\}$。一条路径上包含了多个节点。Q 中记录了各节点到起点 A 的路径长度，其中有一个最短，从优先队列 Q 能快速取出它。

(3) 从优先队列 Q 中弹出最小值，即距离起点 A 最短的节点，这次是 C。在这一步，找到了 A 到 C 的最短路径长度，C 是第 1 个被确定到起点 A 的最短路径的节点。考查 C 的

邻居，其中的新邻居 D、E 直接放入 Q：$\{B_5, D_6, E_7\}$；队列中的旧邻居 B，看经过 C 到 B 是否距离更短，如果是就更新，所以 B_6 更新为 B_5，现在 A 经过 C 到 B，总距离为 5。

(4) 继续从优先队列 Q 中取出距离最短的节点，这次是 B，在这一步，找到了 A 到 B 的最短路径长度，路径是 A-C-B。然后考查 B 的邻居，B 没有新邻居放入 Q；B 在 Q 中的旧邻居 D，通过 B 到它也并没有更近，所以不用更新。Q 现在为 $\{D_6, E_7\}$。

继续以上过程，每个节点都会进入 Q 并弹出，直到 Q 为空时结束。

在优先队列 Q 中找最小值，也就是找距离最短的节点，复杂度为 $O(\log_2 n)$。“BFS+优先队列”求最短路径，算法的**总复杂度**为 $O((n+m)\log_2 n)$，即共检查 $n+m$ 次，每次优先队列复杂度为 $O(\log_2 n)$。

如果不用优先队列，直接在 n 个点中找最小值，复杂度为 $O(n)$，总复杂度为 $O(n^2)$。

$O(n^2)$ 是否一定比 $O((n+m)\log_2 n)$ 好？下面讨论这个问题。

(1) 稀疏图中，点和边的数量差不多，即 $n \approx m$，优先队列的复杂度 $O((n+m)\log_2 n)$ 可以写成 $O(n\log_2 n)$，它比 $O(n^2)$ 好，是非常好的优化。

(2) 稠密图[①]中，点少于边，即 $n < m$ 且 $n^2 \approx m$，优先队列的复杂度 $O((n+m)\log_2 n)$ 可以写成 $O(n^2\log_2 n)$，它比 $O(n^2)$ 差。这种情况下，用优先队列反而不如直接用暴力法。

读者如果学过 Dijkstra 最短路径算法，就会发现，实际上 Dijkstra 算法就是用优先队列实现的 BFS，即 **Dijkstra+优先队列=BFS+优先队列**（队列中存的是从起点到当前点的距离）。

“队列中存的是从起点到当前点的距离”说明了它们的区别，即“Dijkstra+优先队列”和“BFS+优先队列”**并不完全相同**。例如，如果在 BFS 时进入优先队列的是“从当前点到终点的距离”，那么就是贪心最优搜索（Greedy Best First Search），详见 3.8 节“A* 算法”。

根据前面的讨论，Dijkstra 算法也有下面的结论：

(1) 稀疏图，用“Dijkstra+优先队列”，复杂度为 $O((n+m)\log_2 n)=O(n\log_2 n)$；

(2) 稠密图，如果 $n^2 \approx m$，不用优先队列，直接在所有节点中找距离最短的那个点，总复杂度为 $O(n^2)$。

稀疏图的存储用邻接表或链式前向星，稠密图用邻接矩阵。

3. 代码实现

下面用模板题给出 Dijkstra 算法的模板代码。

例 3.20　最短路径[②]

问题描述：给出一个图，求点 1 到其他所有点的最短路径。

输入：第 1 行输入 n 和 m，n 为点的数量，m 为边的数量；第 2～$m+1$ 行中，每行输入 3 个整数 u，v，w，表示 u 和 v 之间存在一条长度为 w 的单向边。$1 \leqslant n \leqslant 3\times10^5$，$1 \leqslant m \leqslant 10^6$，$1 \leqslant u, v \leqslant n$，$1 \leqslant w \leqslant 10^9$。

① 例如全连接图，即所有点之间都有直连的边，V 个点，边的总数 E 为 $(V-1)+(V-2)+\cdots+1 \approx V^2/2=O(V^2)$。

② https://www.lanqiao.cn/problems/1122/learning/

输出：共输出 n 个数，分别表示从 1 点到 1～n 点的最短距离，两两之间用空格隔开。如果无法到达则输出－1。

题目中的 n 很大，路径长度很长，需要用 long long 型。

题目一般不会要求打印路径，因为可能有多条最短路径，不方便系统测试。如果需要打印出最短路径，代码中给出了打印路径的函数 print_path()。

```
#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：节点；n_dis：这个节点到起点的距离
    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];                                       //记录前驱节点
void print_path(int s, int t) {                   //打印从 s 到 t 的最短路径
    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];                                //记录所有节点到起点的距离
bool done[N];                        //done[i]=true 表示到节点 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;                                     //起点到自己的距离为 0
    priority_queue<node> Q;                       //优先队列,存节点信息
    Q.push(node(s, dis[s]));                      //起点进队列
    while(!Q.empty())   {
        node u = Q.top();                         //弹出距起点 s 距离最小的节点 u
        Q.pop();
        if(done[u.id]) continue;       //丢弃已经找到最短路径的节点,即集合 A 中的节点
        done[u.id]= true;
        for(int i=0; i<e[u.id].size(); i++) {   //检查节点 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节点
            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,放入优先队列
                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]);
    }
}
```

扫一扫

视频讲解

3.7 BFS与双端队列

1.2.3节介绍了双端队列，双端队列是一种具有队列和栈性质的数据结构，它能而且只能在两端进行插入和删除。双端队列的经典应用是实现单调队列。下面讲解双端队列在BFS中的应用。

“BFS+双端队列”可以高效率解决一种**特殊图**的最短路径问题：图的边权为0或1。

一般求解最短路径，高效的方法是Dijkstra算法，或者“BFS+优先队列”，复杂度为$O((n+m)\log_2 n)$，n为节点数，m为边数。但是，在边权为0或1的特殊图中，用“BFS+双端队列”可以在$O(n)$时间内求得最短路径。

3.4节介绍了边权为1的情况，是本节边权为0或1的特殊情况。

双端队列的经典应用是单调队列，“BFS+双端队列”的队列也是一个单调队列。

用下面的例题详细解释算法。

例3.21　Switch the lamp on(洛谷 P4667)

时间限制为150ms；内存限制为125.00MB。

问题描述：Casper正在设计电路。有一种正方形的电路元件，在它的两组相对顶点中，有一组会用导线连接起来，另一组则不会。有$N\times M$个这样的元件，排列成N行，每行M个。电源连接到电路板的左上角，灯连接到电路板的右下角。只有在电源和灯之间有一条电线连接的情况下，灯才会亮。为了亮灯，任何数量的电路元件都可以转动90°(两个方向)。

如图 3.11 所示，左图中灯是灭的。在右图中，右数第 2 列的任何一个电路元件被旋转 90°，电源和灯都会连接，灯亮。现在请编写一个程序，求出最小需要旋转多少电路元件。

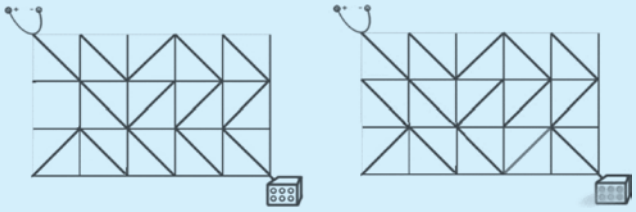


图 3.11 例 3.21 示例图

输入：第 1 行输入两个整数 N 和 M，表示板子的尺寸。在以下 N 行中，每行输入 M 个符号：/或\，表示连接对应电路元件对角线的导线的方向。$1 \leqslant N, M \leqslant 500$。

输出：如果可以亮灯，输出一个整数，表示最少转动电路元件的数量；如果不可能亮灯，输出"NO SOLUTION"。

输入样例：	输出样例：
3 5 \\/\\ \\/// /\\\\	1

本题可以建模为最短路径问题。把起点 s 到终点 t 的路径长度记录为需要旋转的元件数量。从一个点到邻居点，如果元件不旋转，距离为 0；如果需要旋转元件，距离为 1。题目要求找出 s 到 t 的最短路径。样例的网络图如图 3.12 所示，其中实线为 0，虚线为 1。

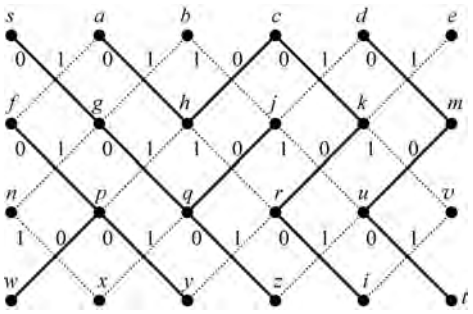


图 3.12 样例的网络图

如果用“BFS＋优先队列”最短路径算法，复杂度为 $O((n+m)\log_2 n)$。题目中节点数 $n=N\times M=250000$，边数 $m=2\times N\times M=500000$，$O((n+m)\log_2 n)\approx 1500$ 万，题目给的时间限制为 150ms，超时。

如果读者透彻理解“BFS＋优先队列”的思想，就能知道优先队列的作用是在队列中找到距离起点最短的那个节点，并弹出它。使用优先队列的原因是，每个节点到起点的距离不同，需要用优先队列排序，找出最小值。

在特殊的情况下，如果不用优先队列，有没有更快的办法找到最小值？本题就是这种特殊情况，边权为 0 或 1。简单地说，就是“边权为 0，插到队头；边权为 1，插入队尾”，这样就**省去了优先队列需要的排序操作**，从而减少了计算，优化了计算复杂度。这个操作用双端队列实现。

下面解释“BFS＋双端队列”计算最短路径的过程。

（1）把起点 s 放入队列。

（2）弹出队头 s。扩展 s 的直连邻居 g，边权为 0 的距离最短，直接插入队头；边权为 1 的直接插入队尾。在样例中，当前队列为$\{g_0\}$，下标记录节点到起点 s 的最短距离。

（3）弹出队头 g_0，扩展它的邻居 b、n、q，现在队列为$\{q_0,b_1,n_1\}$，其中的 q_0，因为边权为 0，直接放入队头。g 被弹出，表示它到 s 的最短路已经找到，后面不再入队。

（4）弹出 q_0，扩展它的邻居 g、j、x、z，现在队列为$\{j_0,z_0,b_1,n_1,x_1\}$，其中 j_0、z_0 边权为 0，直接放入队头。

持续以上过程，直到队列为空。表 3.1 给出了完整的执行过程。

表 3.1　双端队列的执行过程

步骤	出队	邻居	入队	当前队列	最短路径	说　明
1			s	$\{s\}$		
2	s	g	g	$\{g_0\}$	s-s：0	
3	g_0	s、b、n、q	b、n、q	$\{q_0,b_1,n_1\}$	s-g：0	s 已经入过队，不再入队
4	q_0	g、j、x、z	j、x、z	$\{j_0,z_0,b_1,n_1,x_1\}$	s-q：0	g 不再入队
5	j_0	b、d、q、u	d、u	$\{z_0,b_1,n_1,x_1,d_1,u_1\}$	s-j：0	q、b 已经入过队，不再入队
6	z_0	q、u		$\{b_1,n_1,x_1,d_1,u_1\}$	s-z：0	q、u 已经入过队，不再入队
7	b_1	g、j		$\{n_1,x_1,d_1,u_1\}$	s-b：1	g、j 不再入队
8	n_1	g、x		$\{x_1,d_1,u_1\}$	s-n：1	g、x 不再入队
9	x_1	n、q		$\{d_1,u_1\}$	s-x：1	n、q 不再入队
10	d_1	j、m	m	$\{m_1,u_1\}$	s-d：1	m 放在队首，但距离为 1：s-d_1-m_0
11	m_1	d、u		$\{u_1\}$	s-m：1	d、u 不再入队
12	u_1	m、z、j、t	t	$\{t_1\}$	s-u：1	m、z、j 不再入队
13	t_1	u		$\{\}$	s-t：1	队列空，停止

注意以下几个关键点。

（1）如果允许节点多次入队，那么先入队时算出的最短距离大于后入队时算出的最短距离。所以，后入队的节点出队时直接丢弃。当然，最好不允许节点再次入队，在代码中加一个判断即可，代码中的 dis[nx][ny]>dis[u.x][u.y]+d 语句起到了这个作用。

（2）节点出队时，已经得到了它到起点 s 的最短路。

（3）节点进队时，应该计算它到 s 的路径长度再入队。例如，u 出队，它的邻居 v 入队，入队时，v 的距离为 s-u-v，也就是 u 到 s 的最短距离加上(u,v)的边权。

为什么“BFS＋双端队列”的算法过程是正确的？仔细思考可以发现，出队的节点到起点的最短距离是按 0、1、2、…的顺序输出的，也就是说，距离为 0 的节点先输出，然后是距离为 1 的节点…这就是双端队列的作用，它保证距离更近的点总在队列前面，队列是单调的。

因为每个节点只入队和出队一次，且队列内部不需要做排序等操作，所以复杂度为 $O(n)$，n 为节点数量。

代码如下，其中的双端队列用 STL deque 实现。

```
#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记录节点到起点s的最短路径
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){
                //如果一个节点再次入队,那么距离应该更小
                //实际上,由于再次入队时,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。不退出也可以,队列为空自动退出
            }
        }
    }
    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;
}
```

3.8 A* 算 法

扫一扫

视频讲解

A* 搜索算法(A* Search Algorithm，简称 A* 算法)可以高效解决一类最短路径问题：给定一个确定起点、一个确定终点(或者可以预测的终点)，求起点到终点的最短路径。A* 算法常用于最短路径问题的求解，求解最短路径问题的算法很多，如双向广搜的效率也较高，而

A* 算法比双向广搜效率更高。另外，从本节的例题(k 短路径等)可以看出，A* 算法可以解决更复杂的问题。注意，除了图这种应用场合，A* 算法还能在更多场合中得到应用。

A* 算法用于最短路径问题时，可以概括为 A* 算法＝贪心最优搜索＋BFS＋优先队列。在图问题中，“Dijkstra＋优先队列”就是“BFS＋优先队列”，此时也可以把 A* 算法概括为“A* 算法＝贪心最优搜索＋Dijkstra＋优先队列”。

A* 算法的核心是估价函数 $f=g+h$，它的效率取决于 h 函数的设计。

下面的内容，先介绍贪心最优搜索、Dijkstra 算法与 A* 算法的关系，然后推理出 A* 算法的原理和应用。

3.8.1 贪心最优搜索和 Dijkstra 算法

1. 贪心最优搜索

贪心最优搜索(Greedy Best First Search)是一种启发式搜索，效率很高，但是因为使用了贪心的原理，不一定能得到全局最优解。

算法的基本思路是贪心，从起点出发，在它的邻居节点中选择下一个节点时，选择那个到终点最近的节点。当然，实际上不可能提前知道到终点的距离，更不用说挑选出最近的邻居点了。所以，只能采用估计的方法，如在网格图中根据曼哈顿距离估算邻居节点到终点的距离。

如何编程？仍然用“BFS＋优先队列”，不过，进入优先队列的，不是从起点 s 到当前点 i 的距离，而是从当前点 i 到终点 t 的距离。

很明显，贪心最优搜索避开了大量节点，只选择那些“好”节点，速度极快，但是显然得到的路径不一定最优。以边权为 1 的网格图为例，讨论以下两种情况。

(1) 在无障碍的网格图中，贪心最优搜索的结果是最优解。因为用于估算的曼哈顿距离就是实际存在的最短路径，所以每次找到的下一个节点显然是最优的。

(2)在有障碍的网格图中，根据曼哈顿距离选择下一跳节点，路线会一直走到碰壁，然后再绕路，最后得到的不一定是最短路径。

贪心搜索的算法思想是**“只看终点，不管起点”**，走一步看一步，不回头重新选择，走错了也不改正。而且，用曼哈顿距离这种简单的估算，也不能提前绕开障碍。

2. Dijkstra 算法(BFS)

用优先队列实现的 Dijkstra(BFS)算法[①]，能比较高效地求得一个起点到所有其他点的最短路径。Dijkstra 算法有 BFS 的通病：下一步的搜索是盲目的，没有方向感。即使给定了终点，Dijkstra 算法也需把几乎所有的点和边放入优先队列进行处理，直到终点从优先队列弹出为止。所以，它适合用来求一个起点到所有其他节点的最优路径，而不是只求到一个终点的路径。

① 在 3.6 节中已指出“Dijkstra＋优先队列”就是“BFS＋优先队列”。

Dijkstra 的算法思想是**“只看起点，不管终点”**。等把图上的点遍历得差不多了，总会碰巧遇到终点。

3.8.2 A* 算法的原理和复杂度

A* 算法是贪心最优搜索和 Dijkstra 算法的结合，即**“既看起点，又看终点”**。A* 算法比 Dijkstra 算法快，因为它不像 Dijkstra 算法一样盲目。A* 算法比贪心搜索准确，它不仅有贪心搜索的预测能力，而且能得到最优解。

A* 算法是如何结合这两个算法的？

设起点为 s，终点为 t，算法走到当前位置 i 点，把 s-t 的路径分为两部分：s-i-t。

(1) s-i 的路径，由 Dijkstra 算法保证最优性。

(2) i-t 的路径，由贪心搜索进行预测，选择 i 的下一个节点。

(3) 当走到 i 碰壁时，i 将被丢弃，并回退到上一层重新选择新的点 j，j 仍由 Dijkstra 算法保证最优性。

以上思路可以用一个估价函数来具体操作，即

$$f(i)=g(i)+h(i)$$

其中，$f(i)$为对 i 点的评估；$g(i)$为从 s 到 i 的代价；$h(i)$为从 i 到 t 的代价。

若 $g=0$，则 $f=h$，A* 算法就退化为贪心搜索；若 $h=0$，则 $f=g$，A* 算法就退化为 Dijkstra 算法。

A* 算法每次根据最小的 $f(i)$选择下一个点。$g(i)$是已经走过的路径，是已知的；$h(i)$是预测未走过的路径；所以 $f(i)$的性能取决于 $h(i)$的计算。

A* 算法的复杂度，在最差情况下与 Dijkstra(或“BFS+队列”)算法相当，一般情况下会更优。

A* 算法的最终结果是最优的吗？答案是确定的，它的解和 Dijkstra 算法的解一样，是最短路径。当 i 到达终点 t 时，有 $h(t)=0$，那么 $f(t)=g(t)+h(t)=g(t)$，而 $g(t)$是通过 Dijkstra 算法求得的最优解，所以在终点 t 这个位置，A* 算法的解是最优的。

A* 算法用 Dijkstra 算法获得最优性结果；用贪心最优搜索预测扩展方向，减少搜索的节点数量。

3.8.3 3 种算法的对比

图 3.13[①] 准确地对比了 Dijkstra、贪心最优搜索、A* 算法的区别。起点为 s，终点为 t，黑格为障碍。图 3.13 精心设置了障碍的位置，以演示 3 种算法是如何绕过障碍的。

① https://www.redblobgames.com/pathfinding/a-star/introduction.html

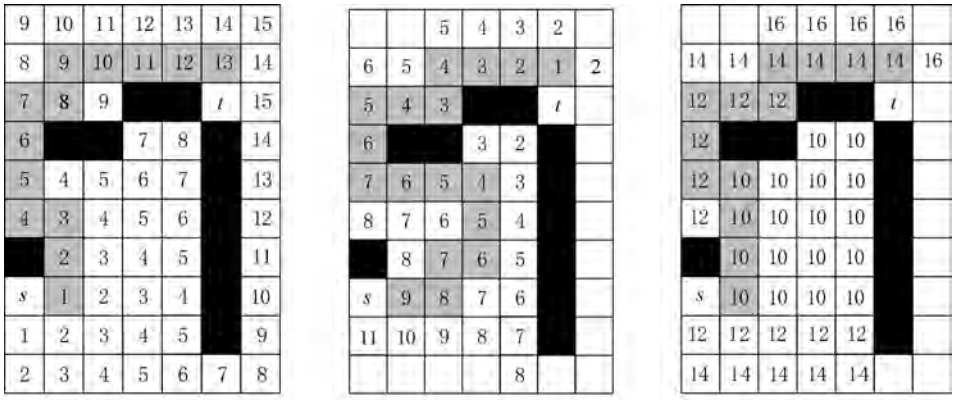


图 3.13　3 种算法对比

从图 3.13 可以比较 3 种算法的计算量。无数字的空白格是算法不需要遍历的格子，空白格越多，计算量越少。Dijkstra 算法遍历了所有的格子，计算量最大；贪心最优搜索的空白格最多，计算量最少；A* 算法计算量居中。

3 种算法都基于“**BFS＋优先队列**”。有数字的格子是搜索过的节点，并进入优先队列处理。灰色阴影格是最后得到的一条完整路径。格子中的数字是距离，按曼哈顿距离计算。

(1) Dijkstra(BFS)算法。格子中的数字是从起点 s 到这个格子的最短距离。算法搜索格子时，把这些格子到起点的距离送入优先队列，当弹出时，就得到了 s 到这些格子的最短路径。最后，当终点 t 从优先队列弹出时，即得到 s 到 t 的最短距离 14。

(2) 贪心最优搜索。格子中的数字是从这个格子到终点 t 的曼哈顿距离。读者可以仔细分析它的工作过程，这里简单说明如下：从 s 沿最小曼哈顿距离一直走到碰壁处的 2；2 从优先队列弹出后，剩下最小的是 3；3 弹出后，剩下最小的是 4…持续这个过程，那些看起来更近但是最终碰壁的节点被逐个弹走，直到拐过障碍，最后到达 t。得到的路径共走了 18 步，不是最优路径。

(3) A* 算法。某个格子 i 中的数字是“s 到 i 的最短路径＋i 到 t 的曼哈顿距离”。算法在扩展格子的过程中，标记数字的格子都会进入优先队列。在图 3.13(c)中，先弹出所有标记为 10 的格子，再弹出标记为 12 的格子，直到最后弹出终点 t。最后得到的 s-t 最短路径也是 14。

如何打印出完整的一条路径？3 个算法都基于 BFS，而 BFS 记录路径是非常简单的：在节点 u 扩展邻居点 v 时，在 v 上记录它的前驱节点 u，即可以从 v 回溯到 u；到达目的后，从终点逐步回溯到起点，就得到了路径。在 Dijkstra 算法中，每次从优先队列中弹出的都是得到了最短路径的节点，从它们扩展出来的邻居节点，也会继续形成最短路径，所以能根据前驱和后继节点的关系方便地打印出一条完整的最短路径。A* 算法用 Dijkstra 算法确定前驱后继的关系，也一样可以打印出一条最短路径。贪心最优搜索的路径打印最简单，就是普通 BFS 的路径打印。

3.8.4 h 函数的设计

在二维平面的图问题中，有以下 3 种方法可以近似计算 h 函数。下面的$(i.x,i.y)$表示 i 点的坐标，$(t.x,t.y)$表示终点 t 的坐标。

(1) 曼哈顿距离。应用场景：只能在 4 个方向(上、下、左、右)移动。

$$h(i)=\text{abs}(i.x-t.x)+\text{abs}(i.y-t.y)$$

(2) 对角线距离。应用场景：可以在 8 个方向上移动，如国际象棋中国王的移动。

$$h(i)=\max\{\text{abs}(i.x-t.x),\text{abs}(i.y-t.y)\}$$

(3) 欧氏距离。应用场景：可以向任何方向移动。

$$h(i)=\text{sqrt}((i.x-t.x)^2+(i.y-t.y)^2)$$

对于非平面问题，需要设计合适的 h 函数，后面的例题中有一些比较复杂的 h 函数。

设计 h 函数时注意以下 3 条基本规则。

(1) g 和 h 应该用同样的计算方法。例如，h 是曼哈顿距离，g 也应该是曼哈顿距离。如果计算方法不同，$f=g+h$ 就没有意义了。

(2) 根据应用情况正确选择 h。各个节点的 h 值应该能正确反映它们到终点的距离远近。例如，下一跳节点有两个选项：$A(280,319)$、$B(300,300)$，如果用曼哈顿距离，应该选 A；用欧氏距离，应该选 B。如果只能走 4 个方向(需要按曼哈顿距离计算路径)，用欧氏距离计算就会出错。

(3) h **应该优于实际存在的所有路径**。前面的例子中，$h(i)$小于或等于 i-t 的所有可能路径长度，也就是说，最后得到的实际路径长度一定大于或等于 $h(i)$。这个规则可以用下面两点讨论来说明。

① $h(i)$比 i-t 的实际存在的最优路径长。假设这条实际的最优路径为 path，由于程序是根据 $h(i)$扩展下一个节点的，所以很可能会放弃 path，而选择另一条非最优的路径，这会造成错误。

② $h(i)$比 i-t 的所有实际存在的路径都短。此时 i-t 上并不存在一条长度为 $h(i)$的路径，如果程序根据 $h(i)$扩展下一节点，最后肯定会碰壁；但是不要紧，程序会利用 BFS 的队列操作弹走这些错误的点，退回到合适的节点，从而扩展出实际的路径，所以仍能保证正确性。

这 3 条基本规则中第 3 点最重要，应用 A* 算法时应特别注意。

A* 算法是“BFS+估价函数”，与之类似，3.9 节的 IDA* 是“DFS+估价函数”。

3.8.5 A* 算法例题

A* 算法的主要难点是设计合适的 h 函数，而编码很容易。例如，图问题中，Dijkstra 算法或 BFS 使用 g 函数，A* 算法使用 $f=g+h$ 函数，那么编码时只要用 f 代替 g 即可。读者可以尝试把图论的最短路径题目改成用 A* 算法实现，如 poj 2243。

下面给出两道复杂一点的例题。

例 3.22　k 短路径问题(poj 2449)

问题描述：给出一个图，包括 n 个点，m 条边。给定起点 s 和终点 t，求从 s 到 t 的第 k 短路径。每个点可以经过两次或两次以上，s 和 t 也可以经过多次。有相同长度的不同路径被视为不同。

输入：第 1 行输入整数 n 和 m，$1\leqslant n\leqslant 1000$，$0\leqslant m\leqslant 1000000$。点从 1 到 n 编号。后面 m 行中，每行输入 3 个整数 u、v、w，表示从点 u 到点 v 的边长为 w。最后一行输入 3 个整数 s、t、k，$1\leqslant s,t\leqslant n$，$1\leqslant k\leqslant 1000$。

输出：输出一个整数，表示第 k 短路径的长度。如果第 k 短路径不存在，输出 -1。

k 短路问题是 A* 算法应用的经典例子，几乎完全套用了 A* 算法的估价函数。下面分别用暴力法和 A* 算法求解。

(1) 暴力法：BFS+优先队列。

用 BFS 搜索所有的路径，用优先队列让路径按从短到长的顺序输出。“BFS+优先队列”求最短路径，其原理是当再次扩展到某个点 i 时，如果这次的新路径比上次到达 i 的路径更短，就替代它；优点队列可以让节点按路径长短先后输出，从而保证最优性。队列的元素是一个二元组$(i,\text{dist}(s\text{-}i))$，即节点 i 和路径 s-i 的长度。

BFS 求所有路径，就是最简单的“BFS+优先队列”，再次扩展邻居 i 时，计算它到 s 的距离，然后直接入队，并不与上次 i 入队的情况进行比较。一个节点 i 会进入优先队列很多次，因为可以从它的多个邻居分别走过来，每次代表了一个从 s 到 i 的路径。优先队列可以让这些路径按 dist 从短到长的顺序输出，i 从优先队列中第 x 次弹出，就是 s 到 i 的第 x 个短路径。对于终点 t，统计它出队列的次数，第 k 次时停止，这就是第 k 短路径。

在 k 短路径问题中，路径有可能形成环路。有的题目允许环路，有的不允许。如果允许环路，那么想在环路上绕多少圈都可以，环路上的节点反复入队，k 可以无限大。在最短路径算法中并不需要判断环路，因为更新操作有去掉环路的隐含作用。

因为暴力法需要生成几乎所有的路径，而路径数量是指数增长的，所以暴力法的复杂度非常高。

下面解释 BFS 暴力搜索所有路径的过程，如图 3.14 所示。

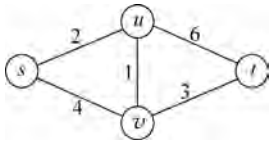


图 3.14　BFS 暴力搜索所有路径的过程

表 3.2 给出了算法的步骤。节点下标表示从 s 到这个节点的路径长度，如 u_2 表示二元组$(u,2)$，即节点 u，以及 s-u 的路径长度 2。步骤中没有列出环路。

表 3.2 k 短路径的队列

步骤	出队	邻居入队	优先队列	新得到的路径	输出队头的路径
1		s	$\{s_0\}$		
2	s_0	u、v	$\{u_2, v_4\}$	s-u_2　　s-v_4	
3	u_2	v、t	$\{v_3, v_4, t_8\}$	s-u_2-v_3　　s-u_2-t_8	s-u_2
4	v_3	t	$\{v_4, t_8, t_6\}$	s-u_2-v_3-t_6	s-u_2-v_3
5	v_4	u、t	$\{t_8, t_6, u_5, t_7\}$	s-v_4-u_5　　s-v_4-t_7	s-v_4
6	u_5	t	$\{t_8, t_6, t_7, t_{11}\}$	s-v_4-u_5-t_{11}	s-v_4-u_5
7	t_6		$\{t_8, t_7, t_{11}\}$		s-u_2-v_3-t_6
8	t_7	u	$\{t_8, t_{11}, u_{13}\}$	s-v_4-t_7-u_{13}	s-v_4-t_7
9	t_8	v	$\{t_{11}, u_{13}, v_{11}\}$	s-u_2-t_8-v_{11}	s-u_2-t_8
10	t_{11}		$\{u_{13}, v_{11}\}$		s-v_4-u_5-t_{11}
11	v_{11}		$\{u_{13}\}$		s-u_2-t_8-v_{11}
12	u_{13}		$\{\}$		s-v_4-t_7-u_{13}

从第 2 列的“出队”可以看到，共产生 10 条路径，按从短到长的顺序排队输出。从起点 s 到终点 t 共有 4 条路径，t 在第 7～10 步出队时，输出了第 1、第 2、第 3、第 4 路径。表 3.2 中也列出了 s 到每个节点的多个路径和它们的长度，如 s-u 有 3 条路径，s-v 有 3 条路径。

(2) A* 算法求 k 短路径问题。

由暴力法可以知道：从优先队列弹出的顺序，是按这些节点到 s 的距离排序的；一个节点 i 从优先队列第 x 次弹出，就是 s-i 的第 x 短路径；终点 t 从队列中第 k 次弹出，就是 s-t 的第 k 短路径。

如何优化暴力法？是否可以套用 A* 算法？

联想前面讲解 A* 算法求最短路径的例子，A* 算法的估价函数 $f(i)=g(i)+h(i)$，g 为从起点 s 到 i 的距离，h 为 i 到终点 t 的最短距离(例子中是曼哈顿距离)。那么在 k 短路径问题中，可以设计几乎一样的估价函数。$g(i)$仍然是起点 s 到 i 的距离；而 $h(i)$只是把曼哈顿距离改为从 i 到 t 的最短距离。这个最短距离如何求？用 Dijkstra 算法，以终点 t 为起点反推，求所有节点到 t 的最短距离即可。

编程非常简单。仍用暴力法的“BFS+优先队列”，但是在优先队列中，用于计算的不再是 $g(i)$，而是 $f(i)$。当终点 t 第 k 次弹出队列时，就是第 k 短路径。

根据前面对 A* 算法原理的解释，求 k 短路径的过程将得到很大优化。虽然在最差情况下，算法复杂度的上界仍是暴力法的复杂度，但优化是很明显的。

poj 2449 的详细代码将在 10.8.3 节给出。

下面再看一道例题。

例 3.23　Power hungry cows(poj 1945)

问题描述：有两个变量 a、b，初始值为 $a=1$，$b=0$。每步可以执行一次 $a\times2$、$b\times2$、$a+b$、$|a-b|$之一的操作，并把结果再存回 a 或 b。问最快多少步能得到一个整数 P？$1\leqslant P\leqslant20000$。

例如，$P=31$，需要 6 步：

	a	b
初始值：	1	0
$a\times2$，存到 b：	1	2
$b\times2$：	1	4
$b\times2$：	1	8
$b\times2$：	1	16
$b\times2$：	1	32
$b-a$：	1	31

输入样例：	输出样例：
31	6

下面是两种解题方法。

(1) BFS+剪枝。本题是典型的 BFS。从$\{a,b\}$可以转移到 8 种情况，如$\{2a,a\}$、$\{2a,b\}$、$\{2b,a\}$、$\{2b,b\}$，等等。把每种$\{a,b\}$看作一个状态，那么一个状态可以转移到 8 个状态。编码时，再加上去重和剪枝。此题 P 不是太大，BFS+剪枝方法可行。

(2) A* 算法。如何设计估价函数 $f(i)=g(i)+h(i)$？$g(i)$为从初始态到 i 状态的步数；$h(i)$为从 i 状态到终点的预期步数，它应该小于实际的步数。如何设计呢？容易观察到，$\{a,b\}$中的较大数，一直乘以 2 递增，是最快的。例如，样例中的 31，在起点状态，$2^5>31$，经 5 步可以超过目标值，所以 $h=5$。

扫一扫

视频讲解

3.9 IDDFS 和 IDA*

迭代加深搜索(Iterative Deepening DFS，IDDFS)是对 BFS 和 DFS 的一个小优化，IDA*(Iterative Deepening A*)是对 IDDFS 的进一步优化。

3.9.1 IDDFS

本节从分析 BFS 和 DFS 的缺点开始，推理出 IDDFS 的优化作用。

1. BFS 和 DFS 的缺点

3.1.5 节曾经提到 BFS 的空间消耗和 DFS 的无效搜索问题，下面先回顾这两个问题。

有这样的应用场景：有一棵树，树非常深；树上的每个点表示解空间的一个状态，问题的解是某个点，现在要求找到这个解。例如，图 3.15 所示为一棵多叉树，黑色星星表示问题的解。

BFS 和 DFS 都能找到这个黑色星星，图 3.16 虚线内是 BFS 和 DFS 遍历的范围。

BFS 的方法是一层一层地逐层遍历，直到在 Depth=2 层找到这个黑色星星。注意在 BFS 的队列中，扩展到黑色星星时，下一层的部分节点也会入队。

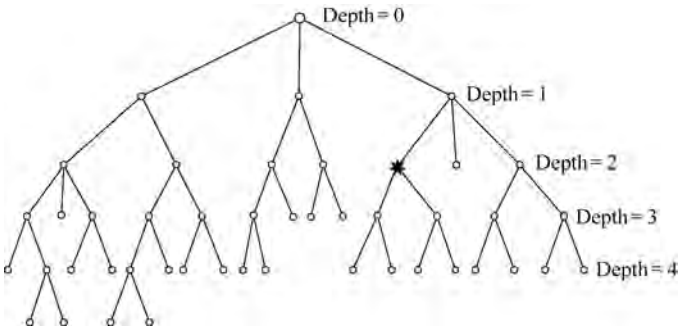


图 3.15 一棵多叉树

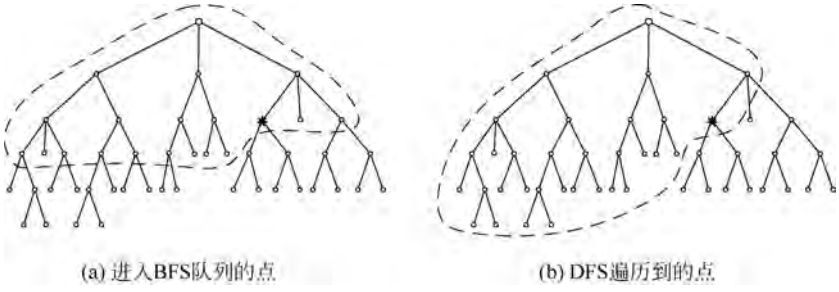


图 3.16 BFS 和 DF 遍历的范围

DFS 若按先左孩子后右孩子的顺序遍历，先遍历完左边所有的子树，再遍历右边子树，直到遇到黑色星星。如果虚线内的树很深，DFS 将遍历到最底层的叶子才会回溯。

BFS 和 DFS 分别有以下缺点。

BFS 的空间消耗：BFS 可能耗费巨大的空间。以二叉树为例，用队列处理节点时，每出队一个节点，就入队两个节点；到第 k 层，队列中就有 2^k 个节点，2^k 是极大的数字，而且每个节点的存储可能需要很大空间。例如，题目常见的内存限制为 64MB，如果每个节点需要 16B，$k=22$ 层时，64MB 空间已经用完了，所以 BFS 只能搜索到 22 层。空间的限制使 BFS 不能用于较深的二叉树。如果能用双向广搜，可以扩大深度，但是如果解的位置不确定，无法以它为起点进行逆向搜索，就不能用双向广搜。

DFS 的无效搜索：DFS 没有 BFS 的空间消耗问题，它只需要能存一条路径的空间即可，即使有 100 层，DFS 的递归深度为 100，也不需要多大的空间。但是 DFS 可能搜索大量无效的节点。在完全二叉树中，DFS 沿着左子树深入，然后逐步回退访问右边的子树。如果解在偏右的子树上，DFS 仍然需要全部检索完左边的子树，才能轮到右边。由于这棵树很深，那么就会搜索左边这些大量无效的节点。例如，100 层的二叉树有 2^{100} 个节点，这是一个天文数字。

2. IDDFS 的原理

如果问题的解在搜索树的较浅层次，IDDFS 可以解决上述问题，它是 BFS 和 DFS 的结合，既不像 BFS 那样浪费空间，也不像 DFS 那样搜索过多无效的节点。

提示 简单地说，IDDFS 是“以 BFS 方式进行 DFS”或“限制深度的 DFS”，代码形式上是 DFS 的，结合了 BFS 的思想。这是一种简单有效的小改进。

IDDFS 的步骤如图 3.17 所示。

(1) 限制深度为 Depth＝0，做第 1 次 DFS。

(2) 限制深度为 Depth＝1，做第 2 次 DFS。

(3) 限制深度为 Depth＝2，做第 3 次 DFS，找到黑色星星，结束。

虚线内部是每次 DFS 遍历到的点。

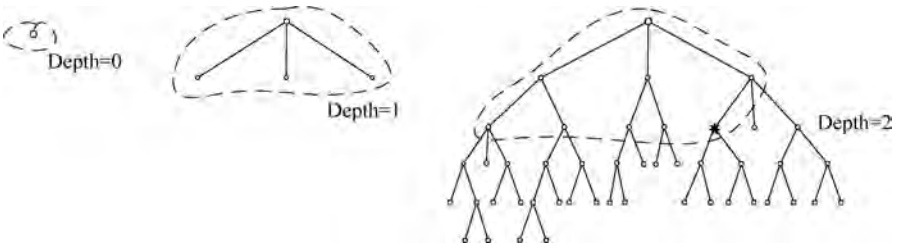


图 3.17　IDDFS 的步骤

算法的伪代码如下。

```
bool IDDFS(s, t, max_depth)            //从 s 出发，如果在深度 max_depth 内找到 t，返回 true
    for d from 0 to max_depth          //逐步扩大 DFS 的深度
       if DFS(s, t, d) == true         //每次扩大深度，都重新从第 1 层开始搜索
           return true
    return false

bool DFS(s, t, d)                      //限制深度为 d 的 DFS
    if (d > depth) return false;       //到达深度 depth，停止并返回 false
    if (s == t) return true;           //找到目标 t，返回 true
    for each adjacent i of s           //对 s 的子节点继续 DFS
        if DFS(i, t, d + 1) == true    //子节点的 DFS 深度为 d - 1
            return true
    return false
```

3. IDDFS 的复杂度

下面分析 IDDFS 的复杂度。

1) 时间复杂度

读者可能发现了 IDDFS 的一个“严重问题”：每次搜索，都要把前几次搜索过的每层都重新再搜索一遍。这是否严重浪费了时间？以 IDDFS 常见的应用背景满二叉树为例，结论是这些重复影响并不大，并没有显著改变复杂度。

第 1 次搜索第 1 层：$2^0 = 2^1 - 1$ 个节点；

第 2 次搜索第 1～2 层：$2^0 + 2^1 = 2^2 - 1$ 个节点；

…

第 k 次搜索第 1～k 层：$2^0+2^1+\cdots+2^k=2^{k+1}-1$ 个节点。

假设在第 k 次搜到了答案，搜索的总数是以上所有步骤的节点数相加，即 $2^1-1+2^2-1+\cdots+2^{k+1}-1\approx 2^{k+2}$。它只比第 k 次搜索多了一倍。每层的节点数是上一层的 2 倍，第 n 层的节点数量是前面所有层的节点数量的总和。在这种指数增长的情况下，第 n 层的计算量与前面所有层的计算量是相同数量级的，也就没有必要节省这些重复的计算。

在二叉树情况下，IDDFS 的时间复杂度为 $O(2^k)$。如果只搜索到第 k 层，BFS 和 DFS 的复杂度也都为 $O(2^k)$，不过根据前面的分析，DFS 搜索的深度远大于 k。

2）空间复杂度

以二叉树为例，IDDFS 搜到第 k 层时，DFS 只需要存 $2k$ 个节点，这比 BFS 要少多了。搜索树每层的分支扩展越多，比 BFS 节省的空间越多。

如表 3.3 所示，IDDFS 的时间复杂度比 DFS 更优，空间复杂度比 BFS 更优。做题时应根据实际情况选择其中的一种搜索方法。

表 3.3 二叉树情况下的复杂度（二叉树共有 n 层，答案在第 k 层）

搜索方法	时间复杂度	空间复杂度	应 用 场 景
DFS	$O(2^n)$	$O(n)$	n 不是很大，即树的深度不是很大
BFS	$O(2^k)$	$O(2^k)$	k 不是很大，即占用的空间没有超过限制
IDDFS	$O(2^k)$	$O(k)$	k 不是很大，且题目对空间限制较大

3.9.2 IDA*

IDDFS 一般需要升级为 IDA*，IDA* 是用估价函数进行剪枝的 IDDFS。

回顾 A* 算法，它相当于“BFS＋估价函数”，估价函数的思想是“前瞻性，能预测”。那么能把估价函数与 DFS 结合起来吗？IDDFS 是一种“盲目”的 DFS 搜索，只是把搜索层次进行了限制。如果在进行 IDDFS 时，能预测出当前继续 DFS 后可能达到的状态，发现不可能到达答案，就直接返回，不再继续深入，从而提高了效率。这个预测是一种剪枝技术。

概括地说，IDA* 是在 IDDFS 中加入估价函数进行剪枝操作的算法。在 IDDFS 的搜索深度限制基础上，用估价函数做**剪枝**操作：如果当前深度＋未来需要的步数＞深度限制，立即返回 false。

下面给出一道 IDA* 例题。

例 3.24 Power calculus（poj 3134）

问题描述：输入正整数 $n(1\leqslant n\leqslant 1000)$，问最少需要几次乘除法可以从 x 得到 x^n，计算过程中 x 的指数要求是正的。例如，为得到 x^{31}，最少乘除 6 次：$x^2=x\times x$，$x^4=x^2\times x^2$，$x^8=x^4\times x^4$，$x^{16}=x^8\times x^8$，$x^{32}=x^{16}\times x^{16}$，$x^{31}=x^{32}\div x$。

输入：每行输入一个整数 n，输入 0 表示结束。

输出：对每个 n 输出一个整数。

对 x 的乘除可以变换为指数的加减，如计算 x^{31}，等于计算 x 的指数 31，$x^2=x\times x$ 指数计算为 $2=1+1$，$x^4=x^2\times x^2$ 为 $4=2+2$，等等。问题转换为：从 1 到 n，只能用加减法，经过多少次运算可以得到 n。

从 1 开始计算，分析一种可能的计算过程，得到图 3.18。根节点的数字 1 位于第 0 层，从 1 往下走的一条路径，就是一个最少的计算次数。例如，最左边的路径，1→2→3→5→…，表示最少用 3 步得到 5，$5=2+3$。对这种路径问题，用 DFS 遍历所有可能的路径，非常合适。本题这种场合非常适合用 IDDFS，答案在较浅的层次。

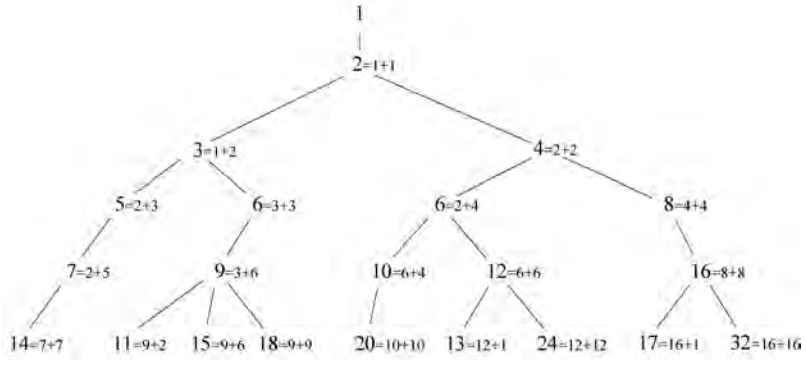


图 3.18　poj 3134 的搜索树（部分）

图 3.18 其实经过了人为极大的简化，节点上很少有重复的数字（只有数字 6 重复了两次）。例如 5，除了走图 3.18 中最左边的路径，还可以走最右边的路径 1→2→4→5，但是图 3.18 中没有画出来。

如果没有人为简化，这棵搜索树将是一棵极为复杂的多叉树，必须进行剪枝。下面代码中的第 9 行用到了一个估价函数进行剪枝。例如图 3.18 中，求 $n=31$，限制搜索深度为 5 层，如果走最左边的路径，走两层后到了“3”这个位置，已经没有必要继续了，因为它最多只能再走 $5-2=3$ 层，最大只能得到 $3\times 2^3=24$，比 31 小。

下面的代码是加上了估价函数的标准 IDDFS 写法。

用 num[]记录 DFS 搜到的一条最短路径，num[i]是路径上第 i 层的数字。可以在第 8 行 if(now==n)中打印出 num[]，观察计算的过程。读者会发现这与图 3.18 有较大不同，因为图 3.18 只画出了部分可能的路径。

```
 1  #include<stdio.h>
 2  #include<string.h>
 3  const int N = 100;                    //最大层次
 4  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
 5  int n, depth;
 6  bool dfs(int now, int d) {            //now 表示当前路径走到的数字,d 表示 now 所在的深度
 7      if (d > depth) return false;      //当前深度大于层数限制
 8      if (now == n)  return true;       //找到目标,注意: 这一句不能放在上一句前面
 9      if (now << (depth - d) < n)       //剪枝: 剩下的层数用最乐观的倍增法也不能达到 n
10          return false;
11      num[d] = now;                     //记录这条路径上第 d 层的数字
12      for(int i = 0; i <= d; i++) {     //遍历之前算过的数,继续下一层
```

```
13          if (dfs(now + num[i], d + 1))       return true;      //加
14          else if (dfs(now - num[i], d + 1)) return true;       //减
15      }
16      return false;
17  }
18  int main() {
19      while(~scanf("%d", &n) && n) {
20          for(depth = 0;; depth++) {          //IDDFS: 每次限制最大搜索 depth 层
21              memset(num, 0, sizeof(num));
22              if (dfs(1, 0))     break;       //从数字 1 开始,当前层为 0
23          }
24          printf("%d\n", depth);
25      }
26      return 0;
27  }
```

【习题】

(1) hdu 1560/1667/4127。

(2) 洛谷 P1032/P2346/P2324/P2534。

小 结

BFS 和 DFS 是很多高级算法的基础,因为它们能遍历出所有状态,从而方便地进行更高级的处理。

本章详解了 BFS、DFS 的原理和各种扩展应用,主要包括以下内容。

(1) 连通性判断。

(2) 剪枝。使用搜索时,一般都需要剪枝,以减少搜索空间,提高效率。

(3) 洪水填充。

(4) BFS 与最短路径。BFS 是最短路径算法的重要技术。

(5) BFS 扩展:双向广搜、优先队列、双端队列。

(6) A* 算法。

(7) IDDFS 和 IDA* 。

BFS 的题目简单一些,DFS 的题目麻烦一些,两种搜索方法都需要大量练习,以达到能不假思索地写出代码的熟练程度。初学者往往难以理解 DFS。DFS 的具体实现是递归,有两个主要步骤:前进、回溯,请通过 3.1 节透彻理解,并通过后面几节的例题和习题进行巩固。