第5章 搜　索

搜索算法是利用计算机的高性能有目的地穷举一个问题的部分或所有的可能情况，从而求出问题的解的一种方法。搜索过程实际上是根据初始条件和扩展规则构造一棵解答树，并寻找符合目标状态的结点的过程。

所有的搜索算法从其最终的算法实现上看，都可以划分成两部分——控制结构和产生系统，而所有算法的优化和改进主要都是通过修改其控制结构完成的。

5.1　基本搜索算法

5.1.1　递归与迭代

递归程序设计是编程语言设计中的一种重要的设计方法，它使许多问题简单化，易于求解。递归的特点：函数或过程直接或间接地调用它本身。通俗来说，递归算法的实质是把问题分解成规模缩小的同类问题的子问题，然后递归调用方法来表示问题的解。递归是一个树结构，字面上可以理解为重复“递推”和“回归”的过程，当“递推”到达底部时就会开始“回归”，其过程相当于树的深度优先遍历。

递归的基本原理：

第一，每一级的函数调用都有自己的变量。

第二，每一次函数调用都会有一次返回。

第三，递归函数中，位于递归调用前的语句和各级被调用函数具有相同的执行顺序。

第四，递归函数中，位于递归调用后的语句的执行顺序和各个被调用函数的顺序相反。

第五，虽然每一级递归都有自己的变量，但是函数代码并不会得到复制。

递归的三大要素：

第一要素，明确这个函数想干什么。先不管函数里面的代码是什么，首先要明白这个函数的功能是什么，要完成什么事。

第二要素，寻找递归结束条件。我们需要找出当参数为什么时，递归结束，之后直接把结果返回。注意，这个时候必须能根据这个参数的值，直接知道函数的结果。

第三要素，找出函数的等价关系式。要不断缩小参数的范围，之后可以通过一些辅助的变量或者操作使原函数的结果不变。

例 5.1 递归求阶乘。

```
#include<stdio.h>
int fact(int);                                  //声明阶乘 fact 函数
int main(){
  int x;
  scanf("%d",&x);
  x = fact(x);                                  //调用 fact 函数返回 int 值
  printf("%d\n",x);
}
int fact(int n){                                //定义阶乘函数
  if(n==1) return 1;                            //若输入的参数是 1,则直接返回 1
  else return n * fact(n-1);                    //递归算法
}
```

所谓迭代，就是在程序中用同一变量存放每次推算出的值，每次循环都执行同一语句，给同一变量赋新值，即用一个新值代替旧值。迭代是一个环结构，从初始状态开始，每次迭代都遍历这个环，并更新状态，多次迭代，直到到达结束状态。理论上递归和迭代的时间复杂度是一样的，但实际应用中（函数调用和函数调用堆栈的开销）递归比迭代的效率要低。

例 5.2 迭代求阶乘。

```
int factorial(int n) {
  if (n == 1) {
  return 1;
  } else {
  return n * factorial(n - 1);
  }
}
```

递归转迭代：

理论上递归和迭代可以相互转换，但实际从算法结构来说，递归声明的结构并不总能转换为迭代结构（原因有待研究）。迭代可以转换为递归，但递归不一定能转换为迭代。

将递归算法转换为非递归算法有两种方法：一种是直接求值（迭代），不需要回溯；另一种是不能直接求值，需要回溯。前者使用一些变量保存中间结果，称为直接转换法；后者使用栈保存中间结果，称为间接转换法。

1. 直接转换法

直接转换法通常用来消除尾递归（tail recursion）和单向递归，将递归结构用迭代结构替代。（单向递归 → 尾递归 → 迭代）

2. 间接转换法

递归实际上利用系统堆栈实现自身调用，我们通过使用栈保存中间结果模拟递归过程，将其转为非递归形式。

由于尾递归函数递归调用返回时正好是函数的结尾，因此递归调用时就不需要保留当前栈帧，可以直接将当前栈帧覆盖掉。

5.1.2 深度优先搜索与广度优先搜索

深度优先搜索与广度优先搜索的控制结构和产生系统很相似，唯一的区别在于对扩展结点的选取。由于其保留了所有的前驱结点，所以在产生后继结点时可以去掉一部分重复的结点，从而提高搜索效率。

这两种算法每次都扩展一个结点的所有子结点，而不同的是，深度优先搜索下一次扩展的是本次扩展出来的子结点中的一个，广度优先搜索扩展的则是本次扩展的结点的兄弟结点。在具体实现上，为了提高效率，所以采用了不同的数据结构。

1. 深度优先搜索

深度优先搜索属于图算法的一种，英文缩写为 DFS，即 Depth First Search。其过程简要来说是对每一个可能的分支路径深入到不能再深入为止，而且每个结点只能访问一次。

举例：小猫爬山。

翰翰和达达饲养了 N 只小猫，这天，小猫们要去爬山。经历千辛万苦，小猫们终于爬上了山顶，但是疲倦的它们再也不想徒步走下山了。

翰翰和达达只好花钱让它们坐索道下山。

索道上的缆车承受的最大质量为 W，而 N 只小猫的质量分别是 $C_1, C_2, \cdots, C_N$。

当然，每辆缆车上小猫的质量之和不能超过 W。

每租用一辆缆车，翰翰和达达就要付 1 元，所以他们想知道，最少需要付多少元才能把这 N 只小猫都运送下山？

输入格式：

第 1 行：包含两个用空格隔开的整数 N 和 W。

第 $2 \sim N+1$ 行：每行一个整数，其中第 $i+1$ 行的整数表示第 i 只小猫的质量 C_i。

输出格式：

输出一个整数，表示最少需要多少元，也就是最少需要多少辆缆车。

数据范围：

$1 \leqslant N \leqslant 18$，

$1 \leqslant C_i \leqslant W \leqslant 108$

输入样例：

```
5 1996
1
2
1994
12
29
```

输出样例：

```
2
```

参考程序：

```
#include <iostream>
#include <algorithm>
```

```
using namespace std;
const int N = 20;
int n, m;
int cat[N], sum[N];                          //cat 表示猫的质量,sum 表示当前车的质量
int ans = N;                                 //ans 表示最优解
void dfs(int u, int k)                       //u 表示为当前第几只猫,k 表示当前车的数量
{
    if(k >= ans) return;
    if(u == n)
    {
        ans = k;
        return;
    }
    for(int i = 0; i < k; i++)
        if(cat[u] + sum[i] <= m)
        {
            sum[i] += cat[u];
            dfs(u + 1, k);
            sum[i] -= cat[u];                //恢复现场
        }
    sum[k] = cat[u];                         //新加一辆车
    dfs(u + 1, k + 1);
    sum[k] = 0;
}
int main()
{
    cin >> n >> m;
    for(int i = 0; i < n; i ++) cin >> cat[i];
    sort(cat, cat + n);                      //从小到大排序
    reverse(cat, cat + n);                   //翻转,从大到小排序
    dfs(0, 0);                               //从第 0 只猫开始考虑,从第 0 辆车开始
    cout << ans << endl;
    return 0;
}
```

2. 广度优先搜索

广度优先搜索算法(又称宽度优先搜索算法)是最简便的图搜索算法之一,英文缩写为BFS,即 Breadth First Search。其过程是在搜索过程中按层次进行搜索,本层的结点没有处理完毕前,不能对下一层结点进行处理,即深度越小的结点越先进行处理。

举例:矩阵距离。

给定一个 N 行 M 列的 01 矩阵 $\boldsymbol{A}$,$\boldsymbol{A}[i][j]$与 $\boldsymbol{A}[k][l]$之间的曼哈顿距离定义为

dist(A[i][j],A[k][l])=|i−k|+|j−l|;

输出一个 N 行 M 列的整数矩阵 $\boldsymbol{B}$,其中:

B[i][j]=min≤x≤N,1≤y≤M,A[x][y]=1 dist(A[i][j],A[x][y]);

输入格式:

第一行两个整数 N,M。

接下来为一个 N 行 M 列的 01 矩阵,数字之间没有空格。

输出格式：

一个 N 行 M 列的矩阵 $\boldsymbol{B}$，相邻两个整数之间用一个空格隔开。

数据范围：

$1 \leqslant N, M \leqslant 1000$

输入样例：

```
3 4
0001
0011
0110
```

输出样例：

```
3 2 1 0
2 1 0 0
1 0 0 1
```

参考程序：

```
#include <iostream>
#include <algorithm>
#include <queue>
#include <cstring>
using namespace std;
typedef pair<int, int>PII;
const int N = 1010;
int n, m;                                          //表示矩阵的长和宽
char g[N][N];                                      //存储矩阵
int d[N][N];                                       //存储每个点到最近 1 的距离
void bfs()
{
    memset(d, -1, sizeof d);                       //d 初始化为-1
    queue<PII> q;
    for(int i = 0; i < n; i ++)
        for(int j = 0; j < m; j++)
            if(g[i][j] == '1')
            {
                q.push({i, j});
                d[i][j] = 0;
            }
    //遍历 4 个方向
    int dx[4] = {-1, 0, 1, 0}, dy[4] = {0, 1, 0, -1};   //按上、右、下、左的顺时针方向
    while (q.size())
    {
        auto t = q.front();
        q.pop();

        int x = t.first, y = t.second;
        for(int i = 0; i < 4; i ++)
        {
            int a = x + dx[i], b = y + dy[i];
            if(a >= 0 && a < n && b >= 0 && b < m && d[a][b] == -1)
```

```
                {
                    d[a][b] = d[x][y] + 1;
                    q.push({a, b});
                }
            }
        }

    }

    int main()
    {
        scanf("%d%d", &n, &m);                              //读入长和宽
        for(int i = 0; i < n; i ++) scanf("%s", g[i]);      //读入矩阵

        bfs();

        for(int i = 0; i < n; i ++)
        {
            for(int j = 0; j < m; j++)
                printf("%d ", d[i][j]);
            puts("");
        }
        return 0;
    }
```

5.1.3 回溯

回溯算法是一种系统地搜索问题解的方法。它的基本思想是：从一条路前走，能进则进，不能进则退回来，换一条路再试。回溯算法是一种通用的解题方法。

应用回溯算法的时候，首先要明确定义问题的解空间。解空间中至少应该包含问题的一个解。确定了解空间后，回溯算法从开始结点出发，以深度优先算法搜索整个解空间。

对于回溯算法，一般采用递归方式实现。

递归函数很容易写：

```
void DFS(int deep)                                  //deep 表示当前递归的深度
{
    if 已经超过递归的深度
return ;
    for 遍历本结点的所有可扩展结点
    {
        DFS(deep + 1);
    }
}
```

当然，回溯算法还可以使用迭代等其他方式来描绘。

5.2 搜索算法的一些优化

5.2.1 剪枝函数

对于回溯算法，我们需要搜索整棵解空间树，剪枝就是通过某种判断，避免一些不必要的遍历过程，剪去解空间树中一些不必要的枝条，从而缩小整个搜索的规模。通常有两种策略来提高回溯算法的搜索效率：一是用约束函数在扩展结点处剪去不满足约束条件的子树；二是用限界函数剪去不能得到最优解的子树。

对于限界函数，一种比较容易的形式是添加一个全局变量记录当前的最优解，而到达结点的时候计算该结点的预期值并与当前的最优解比较。如果不好，则回溯。

5.2.2 双向广度搜索

所谓双向广度搜索，指的是搜索沿正向（从初始结点向目标结点方向搜索）和逆向（从目标结点向初始结点方向搜索）两个方向同时进行，当两个方向上的搜索生成同一子结点时完成搜索过程。

运用双向广度搜索理论上可以减少二分之一的搜索量，从而提高搜索效率。

双向广度搜索一般有两种方法：一种是两个方向交替扩展；另一种是选择结点个数比较少的方向先扩展。显然，第一种方法比第二种方法容易实现，但是由于第二种方法克服了双向搜索中结点生成速度不平衡的状态，因此效率将比第一种方法高。

5.3 实 例 演 示

5.3.1 宝石游戏

问题描述：

宝石游戏非常有趣，它在 13 * 6 的格子里进行。游戏给出红色、蓝色、黄色、橘黄色、绿色和紫色的宝石。当任何三个以上宝石具有相同颜色并且在一条直线（横竖斜）时，这些宝石可以消去游戏截图，如图 5-1 所示。

现在给定当前游戏状态和一组新的石头，请计算当所有石头落下时游戏的状态。

图 5-1 宝石游戏

输入：

第一行 n 表示有 n 组测试数据。

下面每个测试数据包含一个 13 * 6 的字符表，其中 B 表示蓝色，R 表示红色，O 表示橘黄色，Y 表示黄色，G 表示绿色，P 表示紫色，W 表示此处没有宝石。

接下来三行，每行包含一个字符，表示新来的宝石。

最后一行的整数 $m(1 \leqslant m \leqslant 6)$，表示新来宝石的下落位置。

输出：

每个测试样例，输出当所有宝石落下后游戏的状态。

输入样例：

```
1
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
BBWWWW
BBWWWW
OOWWWW
B
B
Y
3
```

输出样例

```
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
WWWWWW
OOYWWW
```

解题思路：

看这个题目可以联想到俄罗斯方块，但是跟俄罗斯方块有些不一样，每个方块都有自己的颜色，需要同色，而且（水平、垂直或对角线方向）连着有三个或三个以上的方块，才能消去。现在的问题是给你一个状态，接下来再给你三个方块，问这三个方块掉下来后最终的状态。需要注意的是，某个方块满足两个或两个以上方向均可消去的情况时要将在这些方向上满足条件的方块都消去，然后下落填补空缺。

可以把整个问题分解成几部分：第一部分是判断当前状态有哪些宝石是可以消去的，如果有这样的宝石，就将它们消去，并在消去的地方用 W 代替，如果没有，那就是游戏的最终状态，输出结果；第二部分是将所有可以下落的宝石下落到无法下落为止，得到一个状态。那么，这个问题就是不断地进行这两步操作，直到得到最终结果。所以，本题的关键是实现这两种操作的两个函数。

任意一块宝石都可以向 8 个方向(实际上只需要 4 个方向)进行扩展,判断是否存在可以消去的宝石。

常见错误:

在没有将当前可以消去的宝石完全消去前,就直接将宝石下落,这样将使得有些可以消去的宝石没有消去,而不可以消去的宝石消去,导致出错。

参考程序:

```
//得到最开始的状态后,按题目指定的位置加入一些宝石
//寻找三个或三个以上相同的宝石,将这些相同的宝石删去后得到新状态
//更新后继续寻找,如此循环,直到没有三个或三个以上相等的宝石为止
#include<iostream>
#include<cstring>
#include<cstdlib>
#define H 13                                    //容器的高度
#define W 6                                     //容器的宽度
#define JH 3                                    //新增宝石的高度
#define JW 1                                    //新增宝石的宽度
using namespace std;
char table[H+2][W+2];                           //容器中宝石的状态
bool isW[H][W];                                 //判断当前格是否为空
char newj[JH][JW];                              //新宝石的状态
bool flag;
const int step[4][2] = {{1,0},{0,1},{1,1},{-1,1}};     //定义 4 个方向的扩展
void downOp()                                   //刷新一遍得到下一个状态(宝石下落后)
{
    int i,j,x;
    for(j=0;j<W;j++)
    {
        x=H-1;
        for(i=H-1;i>=0;i--)                     //宝石下落,依次填满空的位置
        {
            if(isW[i][j])
                continue;
            table[x--][j]=table[i][j];
        }
        for(i=x;i>=0;i--)                       //最后为空的位置赋值为'W'
            table[i][j]='W';
    }
}
void search()                                   //消去宝石
{
    flag = true;                                //默认本次操作没有消去宝石
    int i,j,k,t;
    for(i=0;i<H;i++)
        for(j=0;j<W;j++)
            isW[i][j]=false;

    for(i=0;i<H;i++)                            //开始判断有没有能消去的宝石,若有则消去
    {
```

```
        for(j=0;j<W;j++)
        {
            if(table[i][j]!='W')          //约束判断,若是 W,则无须操作
            {
                for(k=0;k<4;k++)          //向 4 个方向扩展
                {
                    if(i+2*step[k][0]>=0)
                    {                     //约束判断,将不符合条件、越界的宝石舍去
                        if(table[i+step[k][0]][j+step[k][1]] == table[i][j]
                        && table[i+2*step[k][0]][j+2*step[k][1]] == table[i]
[j])
                        {
                            t = 0;
                            while( i+t*step[k][0] >=0
                            && table[i+t*step[k][0]][j+t*step[k][1]] == table
[i][j])
                            {             //找到该方向上的所有能消的宝石
                                isW[i+t*step[k][0]][j+t*step[k][1]] = true;
                                t++;
                            }
                            flag = false;
                        }
                    }
                }
            }
        }
    }
    if(!flag)                             //若有消去的宝石,则刷新状态
        downOp();
}
int main()
{
    int i,j,T,l,h,t;
    cin>>T;
    while(T--)
    {
        for(i=0;i<H;i++)
        {
            cin>>table[i];
        }
        for(i=0;i<JH;i++)
        {
            cin>>newj[i];
        }
        cin>>l;
        l = l-1;
        for(i=H-1;i>=0;i--)
        {
            if(table[i][l]=='W')
            {
```