《算法竞赛入门到进阶》

清华大学出版社　罗勇军　郭卫斌　著

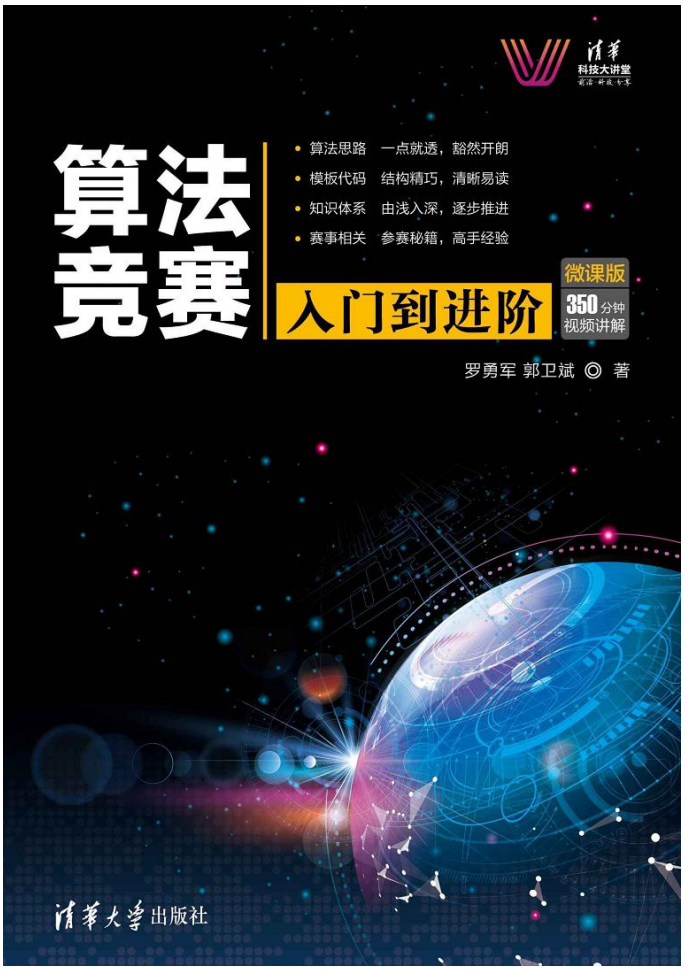
样稿　第 4 章 搜索技术

资料下载有 3 种方式：：

（1）https://github.com/luoyongjun999/code

（2）QQ 群：567554289，群文件

（3）作者 QQ：15512356



目录

第 4 章 搜索技术

- ☑ 递归和排列
- ☑ 子集生成和组合问题
- ☑ BFS 和队列
- ☑ A*算法
- ☑ DFS 和递归
- ☑ 八数码问题
- ☑ 回溯与剪枝
- ☑ 迭代加深搜索
- ☑ IDA*

搜索是基本的编程技术,在算法竞赛学习中是基础的基础。搜索使用的算法是BFS和DFS,BFS 用队列、DFS 用递归来具体实现。在 BFS 和 DFS 的基础上，可以扩展出 A*算法、双向广搜算法、迭代加深搜索、IDA*等技术。本章详细介绍了这些知识点。

搜索技术，是“暴力法”算法思想的具体实现。

人们常说：“要利用计算机强大的计算能力。”如果答案在一大堆数字里面，让计算机一个个去试，符合条件的不就是答案了吗？

没错，最基本的算法思想“暴力法”，就是这样做的。例如，银行卡密码是 6 位数字，共 100 万个，对于计算机来说，尝试 100 万次只需要一瞬间。不过计算机也不是无敌的。为了应对计算机强大的计算能力，可以对密码进行强化设计。比如网络账号密码，大部分网站都要求长度在 8 位以上，并且混合数字、字母、标点等。从 40 多个符号中选 8 个组成密码，数量有 40×39×38×37×36×35×34×33 > 3 万亿，即使用计算机，也不能很快算出来。

暴力法（Brute force，又译为蛮力法）：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。

虽然暴力法常常是低效的代名词，但是它仍然很有用，原因是：

（1）很多问题只能用暴力法解决，比如猜密码。

（2）对于小规模的问题，暴力法完全够用，而且避免了高级算法需要的复杂编码，在竞赛中可以加快解题速度。竞赛中，也可以用暴力法来构造测试数据，以验证高级算法的正确性。

（3）把暴力法当做参照（benchmark）。既然暴力法是“最差”的，那么可以把它当成一个比较，来衡量另外的算法有多“好”。拿到题目后，如果没有其它思路，可以先试试暴力法，看是否能帮助产生灵感。

不过，在具体编程时，常常需要对暴力法进行优化，以减少搜索空间，提高效率。例如利用剪枝技术，跳过不符合要求的情况，从而减少复杂度。

暴力搜索的思路很简单，但是操作起来也并不容易。一般有以下操作：

（1）找到所有可能的数据，并且用数据结构表示和存储。

（2）剪枝。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。

其中的第一步，就可能很不容易。例如迷宫问题，如何列举从起点到终点的所有可能的路径[①]？再例如图论中的“最短路径问题”，在地图上任取两个点，它们之间所有可行的路径，可能是天文数字，以至于根本不能一一列举出来。所以计算最短路径的 Dijkstra 算法，是用贪心法，进行从局部扩散到全局的搜索，不用列举所有可能的路径。

暴力法的主要操作是搜索，搜索的主要技术是 BFS 和 DFS。掌握搜索技术是学习算法竞赛的基础。搜索时，具体的问题会有相应的数据结构，例如队列、栈、图、树等，读者应该能熟练地在这些数据结构上进行搜索的操作。

本章主要讲解 BFS 和 DFS，以及基于它们的优化技术。并以一些经典的搜索问题为例，讲解算法思想，例如排列组合、生成子集、八皇后、八数码、图遍历等等。

4.1 递归和排列

排列和组合问题是暴力枚举的时候常常遇到的。有三种常见情况：

问题 4.1：打印 n 个数的全排列，共 $n!$ 个。

问题 4.2：打印 n 个数中任意 m 个数的全排列，共 $\frac{n!}{(n-m)!}$ 个。

问题 4.3：打印 n 个数中任意 m 个数的组合，共 $C_n^m = \frac{n!}{m!(n-m)!}$ 个。

本节用递归程序来实现问题 4.1 和问题 4.2，问题 4.3 在下一节中讲解。

在计算机编程教材中，都会提到递归的概念和应用。一般都会用数学中的递推方程来讲解递归的概念，例如 f(n) = f(n-1) + f(n-2)。在计算机系统中，递归是通过嵌套来实现的，涉及到指针、地址、栈的使用。

从算法思想上看，递归是把大问题逐步缩小，直到变成最小的同类问题的过程，例如：n → n-1 → n-2 → ... → 1，最后的小问题的解是已知的，一般是给定的初始条件。在递归的过程中，由于大问题和小问题的解决方法完全一样，那么自然可以想到，大问题的程序和小问题的程序可以写成一样。一个递归函数，直接调用自己，就实现了程序的复用。

递归和分治法的思路非常相似，分治是把一个大问题分解为多个类型相同的子问题。事实上，一些涉及分治法的问题可以用递归来编程，典型的有快速排序、归并排序等。

对于编程初学者来说，递归是一个难以理解的编程概念，很容易把初学者绕晕。为了帮

① 用 DFS 可以实现，程序也非常短。学完本章后，读者就能轻松地写出程序。

助理解，可以一步步打印出递归函数的输出，看它从大到小解决问题的过程。

编程竞赛中的暴力法，常常需要考虑所有可能的情况。用递归编程，可以轻松方便地实现对搜索空间所有状态的遍历。

【问题 4.1】打印 n 个数的全排列

在用递归解决这个问题之前，先给出 STL 的实现方法。

1. 用 STL 输出全排列

如果需要全排列的场景比较简单，可以直接用 C++ STL 的库函数 next_permutation()。它按字典序输出下一个排列。在使用之前，先用 sort 给数据排序，得到最小排列；然后每调用 next_permutation()一次，就得到一个大一点的排列。

next_permutation()的优点是能按从小到大的顺序输出排列。

```
#include<iostream>
#include<algorithm>                    //包含 sort 和 next_permutation 函数
using namespace std;
int main(){
    int data[4] = {5, 2, 1, 4};
    sort(data, data + 4);               //排序，得到最小排列
    do{
        for(int i = 0; i < 4; ++i)      //输出一个排列
            cout << data[i] << " ";
        cout << endl;
    }while(next_permutation(data, data + 4)); //把下一个排列放在 data 中
    return 0;
}
```

2. 用递归求全排列

下面用递归求全排列，代码很短，但是理解起来并不容易。读者可以自己打印每一个全排列的输出，然后认真理解。

在用递归之前，为了对比，先给出一个简单粗暴的方法：以 10 个数的全排列为例，用排列组合的思路，写一个 10 级的 for 循环，在每个 for 中，选一个和前面的 for 用过的都不同的数。当 n = 10 时，一共有 10！= 3,628,800 个排列。

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

```
int data[]={7,1,2,3,4,5,6,8,9,10,12};        //本例子中，用到前 10 个数
int main(){
    int num = 10;
    int i, j, k, m, n, p, q, r, s, t;        //10 个 for 循环
    for(i = 0; i<num; i++)
        for(j=0; j<num; j++)
            if(j != i)                                  //让 j 不等于 i
                for(k = 0; k < num; k++)
                  if(k != j && k != i)                //让 k 不等于 i, j
                      for(m = 0; m < num; m++)
                          if(m!=j && m!=i && m!=k) //让 m 不等于 i, j, k
                              ......
                  //最后，打印出一个全排列：cout<<data[i]<<data[j]......
}
```

上述的程序看起来很“笨”。下面用递归来写，显得很“美”。

递归求全排列的思路：设定数字是{1 2 3 4 5.....n}

（1）让第一个数不同，得到 n 个数列。其办法是：把第 1 个和后面每个数交换即可。

1 2 3 4 5.....n

2 1 3 4 5.....n

.....

n 2 3 4 5.....1

以上 n 个数列，只要第一个数不同，不管后面 n-1 个数是怎么排列的，这 n 个数列都不同。

这是递归的第一层。

（2）继续：在上面的每个数列中，去掉第一个数，对后面的 n-1 个数进行类似的排列。例如从上面第 2 行的{2 1 3 4 5.....n}进入第二层（去掉首位 2）：

1 3 4 5.....n

3 1 4 5.....n

.....

n 3 4 5.....1

以上 n-1 个数列，只要第一个数不同，不管后面 n-2 个数是怎么排列的，这 n-1 个数列都不同。

这是递归的第二层。

（3）重复以上步骤，直到用完所有数字。

上面所有过程完成后，数列的总个数是：n*(n-1)*(n-2).....*1 = n!

递归打印全排列

```
#include<bits/stdc++.h>
using namespace std;
#define Swap(a, b) {int temp = a; a = b; b = temp;}
            //交换。也可以直接用 C++ STL 中的 swap()函数，但是速度慢一些
int data[]={1,2,3,4,5,6,8,9,10,32,15,18,33};//本例子只用到前面 10 个数
int num = 0;                //统计全排列的个数，验证是不是 3628800
int Perm(int begin, int end){
    int i;
    if(begin == end) {    // 递归结束，产生一个全排列
                            //如果有必要，在这里打印或处理这个全排列
         num++;             //统计全排列的个数
     }
    else
        for(i = begin; i <= end; i++) {
              Swap(data[begin], data[i]);//把当前第 1 个数与后面所有数交换位置
              Perm(begin+1, end);
              Swap(data[begin], data[i]);  //恢复，用于下一次交换
        }
}
int main(){
    Perm(0, 9);                    //求 10 个数的全排列
    cout<<num<<endl;               //打印出排列总数，num = 10! =3628800
}
```

用这个程序可以检验普通电脑的计算能力。在上面的程序中加入 clock()统计时间：

```
#include <ctime>
int main() {
    clock_t start, end;
    start = clock();
    Perm(0, 9);
    end = clock();
    cout << (double)(end - start) / CLOCKS_PER_SEC << endl;
}
```

在作者的笔记本电脑上运行上述程序：

1、Perm(0, 9)，计算 10 个数的全排列：10! = 3,628,800。用时 0.055s。

2、Perm(0,10)，计算 11 个数的全排列：11! = 39,916,800。用时 0.598s。

3、Perm(0,11)，计算 12 个数的全排列：12! = 479,001,600。用时 7.305s。

12!/11!/10!的比值，与 7.305s/0.598s/0.055s 的比值非常接近。

结论：笔记本电脑的计算能力大约是每秒千万次数量级。

竞赛题在一般情况下限时 1s。所以对于需要全排列的题目，其元素个数应该少于 11 个。

需要注意的是，从算法复杂度上看，上述 2 个程序，复杂度一样，都是 O(n!)。对求全排列这样的问题，不可能有复杂度小于 O(n!)的算法，因为输出的数量就是 n!。在算法理论中，对必须要输出的元素进行的计数，叫做“平凡下界”，这是程序运行所需要的最少花费。

上面的程序只要进行小的修改，就能解决问题 4.2。

【问题 4.2】打印 n 个数中任意 m 个数的全排列

例如在 10 个数中取任意 3 个数的全排列，在 Perm()中只修改一个地方就可以了：

```
if(begin == 3) {                    // 把 Perm()函数中的 end 改为 3 即可，其它都不变
    cout<<data[0]<<data[1]<<data[2]<<endl; //打印 10 个数中 3 个数的全排列
    num++;                          //统计全排列的个数，应该是 10*9*8 = 720 个
}
```

【问题 4.3】打印 n 个数中任意 m 个数的组合

问题 4.3 和问题 4.1 的区别是：排列是有序的，组合是无序的。其中一个特例是：n 个数取 n 个数的组合，只有 1 种情况，就是这 n 个数本身。

问题 4.3 在下一节“子集生成”中讲解。

4.2 子集生成和组合问题

在上节求 10 个数的排列问题中，如果并不需要输出全排列，而是组合，即子集（子集内部的元素是没有顺序的），那么该如何做呢？

一个包含 n 个元素的集合{$a_0, a_1, a_2, a_3, ..., a_{n-1}$}，它的子集有{$\phi$}，{$a_0$}，{$a_1$}，{$a_2$}, ..., {$a_0, a_1, a_2$}, ..., {$a_0, a_1, a_2, a_3, ..., a_{n-1}$}，共 2^n 个。

用二进制的概念进行对照是最直观的。

例如 n=3 的集合{a_0, a_1, a_2}，它的子集和二进制数的对应关系是：

子集	ϕ	a_0	a_1	a_1,a_0	a_2	a_2,a_0	a_2,a_1	a_2,a_1,a_0
二进制数	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1

所以，每个子集对应了一个二进制数；这个二进制数中的每个 1，都对应了这个子集中的某个元素。而且，子集中的元素，是没有顺序的。

从这个表也可以理解，为什么子集的数量是 2^n 个，因为所有二进制数的总个数是 2^n。

下面的程序，通过处理每个二进制数中的 1，打印出了所有的子集。

```
#include<bits/stdc++.h>
using namespace std;
void print_subset(int n){
     for(int i=0;i<(1<<n);i++) {
      //i：0~2^n，每个 i 的二进制数对应一个子集。一次打印一个子集，最后得到所有子集
        for(int j=0;j<n;j++)//打印一个子集，即打印 i 的二进制数中所有的 1
           if(i & (1<<j))    //从 i 的最低位开始，逐个检查每一位，如果是 1，打印
               cout<<j<<" ";
        cout<<endl;
    }
}
int main(){
    int n;
    cin >> n;                // n：集合中元素的总数量
    print_subset(n);        // 打印所有的子集
}
```

回到上节的问题 4.3：打印 n 个数中任意 m 个数的组合。对照子集生成的二进制方法，已经知道，一个子集对应一个二进制数；那么一个有 k 个元素的子集，它对应的二进制数中有 k 个 1。所以，问题就转化为：查找 1 的个数为 k 个的二进制数，这些二进制数就是需要打印的子集。

如何判断二进制数中 1 的个数为 k 个[①]？简单的方法是对这个 n 位的二进制数逐位检查，共需要检查 n 次。

有一个更快的方法，它可以直接定位二进制数中 1 的位置，跳过中间的 0。它用到一个神奇的操作：kk = kk & (kk - 1)，功能是消除 kk 的二进制数的最后一个 1。连续进行这个操作，每次消除一个 1，直到全部消除，操作次数就是 1 的个数。例如二进制数 1011，经过连续 3 次操作后，所有 1 都消除了：

1011 & (1011 - 1) = 1011 & 1010 = 1010

1010 & (1010 - 1) = 1010 & 1001 = 1000

1000 & (1000 - 1) = 1000 & 0111 = 0000

① glibc 有处理二进制数的内部函数，其中 int __builtin_popcount (unsigned int x)，直接返回 x 中 1 的个数。

利用这个操作，可以计算出二进制数中 1 的个数。用 num 统计 1 的个数，具体步骤是：

（1）用 kk = kk & (kk - 1)清除 kk 的最后一个 1；

（2）num++；

（3）继续上述操作，直到 kk = 0。

在树状数组中，也有一个类似的操作：lowbit(x) = x & -x，功能是计算 x 的二进制数的最后一个 1。

下面的程序在子集生成程序的基础上实现了问题 4.3 的要求：

```
#include<bits/stdc++.h>
using namespace std;
void print_set(int n,int k){
     for(int i = 0; i < (1<<n); i++){
        int num = 0, kk = i;     //num 统计 i 中 1 的个数；kk 用来处理 i
        while(kk){
            kk = kk&(kk-1);      //清除 kk 中最后一个 1
            num++;                 //统计 1 的个数
        }
        if(num == k){            //二进制数中的 1 有 k 个，符合条件
            for(int j = 0; j < n; j++)
                if(i & (1<<j))
                    cout << j << " ";
            cout << endl;
        }
    }
}
int main(){
    int n, k;                    // n：元素的总数量。k：个数为 k 的子集
    cin >> n >> k;
    print_set(n,k);
}
```

4.3 BFS

4.3.1 BFS 和队列

深度优先搜索（DFS, Depth-First Search）和宽度优先搜索（BFS, Breadth-First Search，或称为广度优先搜索）是基本的暴力技术，常用于解决图、树的遍历问题。

先了解算法的思路。以老鼠走迷宫为例，这是 DFS 和 BFS 在现实中的模型。迷宫内部的路错综复杂，老鼠从入口进去后，怎么才能找到出口？有两种不同的方法：

（1）一只老鼠走迷宫。它在每个路口，都选择先走右边（当然，选择先走左边也可以），能走多远就走多远；直到碰壁无法再继续往前走，然后往回退一步，这一次走左边，然后继续往下走。用这个办法，能走遍**所有**的路，而且**不会重复**（这里规定回退不算重复走）。这个思路，就是 DFS。

（2）一群老鼠走迷宫。假设老鼠是无限多的，这群老鼠进去后，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。很显然，**所有**的道路都会走到，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，所以，可以把 BFS 看成是并行计算的模拟。

具体编程时，一般是用队列这种数据结构来具体实现 BFS，甚至可以说“BFS = 队列”；对于 DFS，也可以说“DFS = 递归”，因为用递归实现 DFS 是最普遍的。DFS 也可以用栈这种数据结构来直接实现，栈和递归在算法思想上是一致的。

本节先讲 BFS。下面用一个图遍历的题目来介绍 BFS 和队列。

hdu 1312　Red and Black

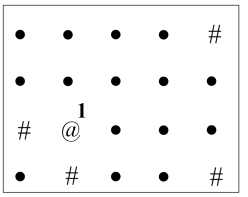
有一个长方形的房间，铺着方形瓷砖，每块瓷砖都是红色或黑色。一个人站在黑色的瓷砖上，他可以按上、下、左、右方向移动到相邻的瓷砖。但他不能在红瓦上移动，他只能在黑瓦上移动。编程计算他可以达到的黑色瓷砖的数量。

Input：第一行包含两个正整数 W 和 H，W 和 H 分别表示 x 方向和 y 方向上的瓦片数量。W 和 H 均不超过 20。下面有 H 行，每个行包含 W 个字符。每个字符表示一个瓦片的颜色。用符号表示如下：'•'表示黑色瓦片；'#'表示红色瓦片；'@'代表黑瓦片上的人，在数据集中只出现一次。

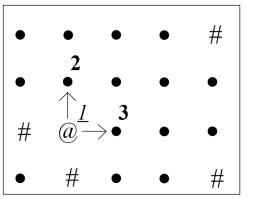
Output：一个数字，他从初始砖块能到达的砖块总数量（包括起点）。

这个题目跟老鼠走迷宫差不多：'#'相当于不能走的陷阱或墙壁，'•'是可以走的路。下面按“一群老鼠走迷宫”的思路编程。

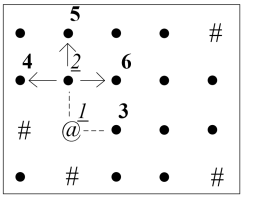
要遍历所有可能的点，可以这样走：从起点 1 出发，走到它所有的邻居 2、3；逐一处理每个邻居，例如在邻居 2 上，再走它的所有邻居 4、5、6；继续以上过程，直到所有点都被走到。这是一个“扩散”的过程，如果把搜索空间看成一个池塘，丢一颗石头到起点位置，激起的波浪会一层层扩散到整个空间。需要注意的是，扩散按从近到远的顺序进行，因此，每个被扩散到的点，从它到起点的路径都是最短的。这个特征，对解决迷宫这样的最短路径问题很有用。



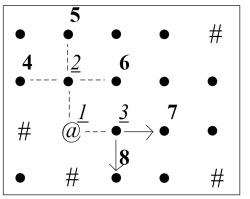
(a) 1 进队列



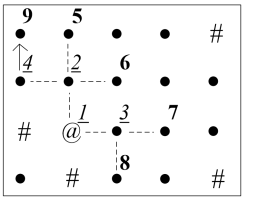
(b) 1 出队；2, 3 进队



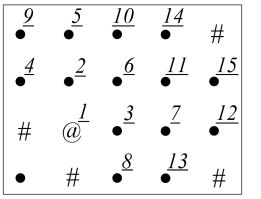
(c) 2 出；4, 5, 6 进



(d) 3 出；7, 8 进



(e) 4 出；9 进



(k)最后结果

图 4.1 BFS 过程

用队列来处理这个扩散过程，非常清晰易懂，对照上图：

(a) 1 进队列。当前队列是{1}。

(b) 1 出队，1 的邻居 2, 3 进队。当前队列{2, 3}。（可以理解为：从 1 扩散到 2、3。）

(c) 2 出队，2 的邻居 4, 5, 6 进队。当前队列{ 3, 4, 5, 6}。（从 2 扩散到 4、5、6。）

(d) 3 出队，7, 8 进。当前队列{ 4, 5, 6, 7, 8}。（从 3 扩散到 7、8。）

(e) 4 出队，9 进。当前队列{5, 6, 7, 8, 9}。

(f) 5 出队，10 进。当前队列{6, 7, 8, 9, 10}。

(g) 6 出队，11 进。当前队列{7, 8, 9, 10, 11}。

(h) 7 出队，12, 13 进。当前队列{8, 9, 10, 11, 12, 13}。

(i) 8, 9 出队；10 出队，14 进。当前队列{11, 12, 13, 14}。

(j) 11 出队，15 进。当前队列{12, 13, 14, 15}。

(k) 12, 13, 14, 15 出队。当前队列是空{}，结束。

hdu 1312 题的 BFS 程序

```
#include<bits/stdc++.h>
using namespace std;
char room[23][23];
int dir[4][2] = {
    {-1,0},  //向左。左上角坐标是(0, 0)
    {0,-1},  //向上
    {1,0},   //向右
    {0,1}    //向下
};
```

```
int Wx, Hy, num;                    //Wx 行，Hy 列。用 num 统计可走的位置有多少
#define CHECK(x, y) (x<Wx && x>=0 && y >=0 && y<Hy) //是否在 room 里
struct node {int x,y;};
void BFS(int dx,int dy){
     num=1;                          //起点也包含在砖块内
     queue <node> q;                 //队列中放坐标点
     node start, next;
     start.x = dx;
     start.y = dy;
     q.push(start);
     while(!q.empty()) {
        start = q.front();
        q.pop();
        //cout<<"out"<<start.x<<start.y<<endl;    //打印出队列情况，进行验证
        for(int i=0; i<4; i++) { //按左、上、右、下，4 个方向顺时针逐一搜索
            next.x = start.x + dir[i][0];
            next.y = start.y + dir[i][1];
            if(CHECK(next.x,next.y) && room[next.x][next.y]=='.') {
                room[next.x][next.y]='#';          //进队之后，标记为已经处理过
                num++;
                q.push(next);
            }
        }
    }
}
int main(){
    int x, y, dx, dy;
    while (cin >> Wx >> Hy) {              //Wx 行，Hy 列
        if (Wx==0 && Hy==0)                //结束
            break;
        for (y = 0; y < Hy; y++) {         //有 Hy 列
            for (x = 0; x < Wx; x++) {     //一次读入一行
                cin >> room[x][y];
                if(room[x][y] == '@') {    //读入起点
                    dx = x;
                    dy = y;
```

```
                }
            }
        }
        num = 0;
        BFS(dx, dy);
        cout << num << endl;
    }
    return 0;
}
```

【习题】

poj 3278 Catch That Cow
poj 1426 Find The Multiple
poj 3126 Prime Path
poj 3414 Pots
hdu 1240 Asteroids!
hdu 4460 Friend Chains

4.3.2 八数码问题和状态图搜索

BFS 搜索处理的对象，不仅可以是一个数，还可以是一种“状态”。八数码问题是典型的状态图搜索问题。

1. 八数码问题

在一个 3×3 的棋盘上，放置编号为 1~8 的 8 个方块，每个占一格，另外还有一个空格。与空格相邻的数字方块可以移动到空格里。任务 1：指定初始棋局和目标棋局，计算出最少的移动步数；任务 2：输出数码的移动序列。

1	2	3
	8	4
7	6	5

1		3
8	2	4
7	6	5

图 4.2 初始棋局和目标棋局

把空格看成 0，一共有 9 个数字。样例输入：

1 2 3 0 8 4 7 6 5

1 0 3 8 2 4 7 6 5

样例输出：2

把一个棋局看成一个状态图，总共有 9! = 362880 个状态。从初始棋局开始，每次移动转到下一个状态，到达目标棋局后停止。

八数码问题是一个经典的 BFS 问题。前面章节提到 BFS 是从近到远的扩散过程，适合解决最短距离问题。八数码从初始状态出发，每次转移都逐步逼近目标状态。转移一次，步数加一，到达目标时，经过的步数就是最短路径。

下图是样例的转移过程。图中起点为(A, 0)，A 表示状态，即{1 2 3 0 8 4 7 6 5}这个棋局；0 是距离起点的步数。从初始状态 A 出发，移动数字 0 到邻居位置，按左、上、右、下顺时针顺序，有 3 个转移状态 B、C、D；目标状态是 F，停止。

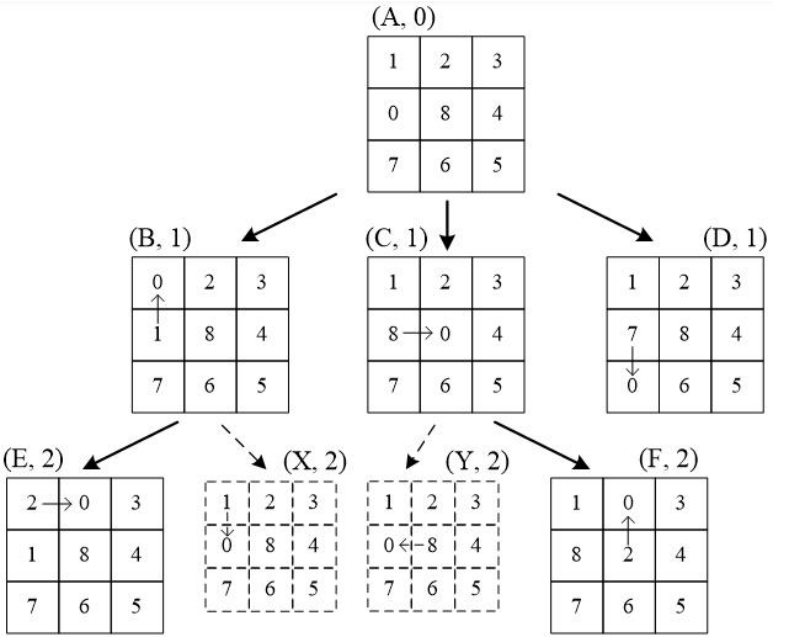


图 4.3 八数码问题的搜索树

用队列描述这个 BFS 过程：

(1) A 进队列，当前队列是{A}；

(2) A 出队，A 的邻居 B、C、D 进队列，当前队列是{B, C, D}，步数为 1；

(3) B 出队，E 进队，当前队列是{C, D, E}，E 的步数为 2；

(4) C 出队，转移到 F，检验 F 是目标状态，停止，输出 F 的步数 2。

仔细分析上述过程，发现从 B 状态出发，实际上有 E、X 两个转移方向，而 X 正好是初始态 A，重复了。同理 Y 状态也是重复的。如果不去掉这些重复的状态，程序会产生很多无效操作，复杂度大大增加。因此，八数码的重要问题其实是判重。

如果用暴力的方法判重，每次把新状态与全部 9! = 362880 个状态对比，可能有 9! × 9! 次检查，不可行。 需要一个快速的判重方法。

本题可以用数学方法“康托展开(cantor expansion)”来判重。

2. 康托展开

康托展开是一种特殊的哈希函数。在本题中，康托展开完成了这样一个工作：

状态	012345678	012345687	012345768	012345786	.....	876543210
Cantor	0	1	2	3	.....	362880-1

第一行是 0~8 这 9 个数字的全排列，共 9! = 362880 个，按从小到大排序。第二行是每个排列对应的位置，例如最小的{012345678}在第 0 个位置，最大的{876543210}在最后的 362880-1 这个位置。

函数 Cantor ()实现的功能是：输入一个排列，即第一行某个排列；函数计算出它的 Cantor 值，即第二行对应的数。

Cantor ()的复杂度为 $O(n^2)$，n 是集合中元素的个数。在本题中，完成搜索和判重的总复杂度是 $O(n!n^2)$，远比用暴力判重的总复杂度 O(n!n!)小。

有了这个函数，八数码的程序能很快判重：每转移到一个新状态，就用 Cantor ()判断这个状态是否处理过，如果处理过，则不转移。

下面举例讲解康托展开的**原理**。

例子：判断 2143 是{1, 2, 3, 4}的全排列中第几大的数。

计算排在 2143 前面的排列数目，可将问题转换为以下排列的和：

（1）首位小于 2 的所有排列。比 2 小的只有 1 一个数，后面 3 个数的排列有 3×2×1 = 3!个（即：1234, 1243, 1324, 1342, 1423, 1432）。写成 1×3! = 6。

（2）首位为 2、第二位小于 1 的所有排列。无，写成 0×2! = 0。

（3）前两位为 21、第 3 位小于 4 的所有排列。只有 3 一个数（即 2134），写成 1×1! = 1。

（4）前 3 位为 214、第 4 位小于 3 的所有排列。无，写成 0×0! = 0。

求和：1×3! + 0×2! + 1×1! + 0×0! = 7，所以 2143 是第 8 大的数。如果用 int visited[24]数组记录各排列的位置，{2143}就是 visited[7]；第一次访问这个排列时，置 visited[7] = 1；再次访问这个排列的时候，发现 visited[7]等于 1，说明已经处理过，判重。

根据上面的例子，得到康托展开公式：

把一个集合产生的全排列，按字典序排序，第 X 个排列的计算公式：

X = a[n]×(n-1)! + a[n-1]×(n-2)! + ... +a[i]×(i-1)! + ... + a[2]×1! + a[1]×0!

其中 a[i]表示原数的第 i 位在当前未出现的元素中排在第几个（从 0 开始），并且有 0 <= a[i] < i ，1 <= i <= n。

上述过程的反过程是康托逆展开：某个集合的全排列，输入一个数字 k，返回第 k 大的排列。

下面的程序用“BFS + Cantor”解决了八数码问题，其中 BFS 用 STL 的 queue 实现[①]。

---

① 本题中的队列，由于比较简单，如果不用 STL，也可以用简单的方法模拟队列，请搜索网上的代码。

```
#include<bits/stdc++.h>
const int LEN = 362888;        //状态共 9!=362880 种
using namespace std;
struct node{
    int state[9];       //记录一个八数码的排列，即一个状态
    int dis;             //记录到起点的距离
};

int dir[4][2] = {{-1,0}, {0,-1},{1,0},{0,1}};
                          //左、上、右、下顺时针方向。左上角坐标是(0,0)
int visited[LEN]={0};  //与每个状态对应的记录，Cantor 函数对它置数，并判重
int start[9];            //开始状态
int goal[9];             //目标状态
long int factory[] = {1,1,2,6,24,120,720,5040,40320,362880};
                            //Cantor 用到的常数
bool Cantor(int str[], int n) {     //用康托展开判重
     long result = 0;
     for(int i = 0; i < n; i++) {
        int counted = 0;
        for(int j = i+1; j < n; j++) {
            if(str[i] > str[j])       //当前未出现的元素中是排在第几个
                ++counted;
        }
        result += counted*factory[n-i-1];
    }
    if(!visited[result]) {            //没有被访问过
        visited[result] = 1;
        return 1;
    }
    else
        return 0;
}
int bfs() {
    node head;
    memcpy(head.state, start, sizeof(head.state));  //复制起点的状态
    head.dis = 0;
```

```
    queue <node> q;           //队列中放状态
    Cantor(head.state, 9);  //用康托展开判重，目的是对起点的 visited[]赋初值
    q.push(head);             //第一个进队列的是起点状态

    while(!q.empty()) {              //处理队列
        head = q.front();
        if(memcmp(head.state,goal,sizeof(goal)) == 0)   //与目标状态对比
                    return head.dis;              //到达目标状态，返回距离，结束
        q.pop();                         //可在此处打印 head.state，看弹出队列的情况
        int z;
        for(z = 0; z < 9; z++)         //找这个状态中元素 0 的位置
            if(head.state[z] == 0)     //找到了
                break;
            //转化为二维，左上角是原点(0, 0)。
        int x = z%3;          //横坐标
        int y = z/3;          //纵坐标
        for(int i = 0; i < 4; i++){   //上、下、左、右最多可能有 4 个新状态
            int newx = x+dir[i][0];    //元素 0 转移后的新坐标
            int newy = y+dir[i][1];
            int nz = newx + 3*newy;    //转化为一维
            if(newx>=0 && newx<3 && newy>=0 && newy<3) {//未越界
                node newnode;
                memcpy(&newnode,&head,sizeof(struct node));//复制这新的状态
                swap(newnode.state[z], newnode.state[nz]);//把 0 移动到新的位置
                newnode.dis ++;
                if(Cantor(newnode.state, 9))          //用康托展开判重
                    q.push(newnode);                    //把新的状态放进队列
             }
        }
    }
    return -1;            //没找到
}
int main(){
    for(int i = 0; i < 9; i++)  cin >> start[i];       //初始状态
    for(int i = 0; i < 9; i++)  cin >> goal[i];        //目标状态
    int num = bfs();
```

```
    if(num != -1)  cout << num << endl;
    else           cout << "Impossible" << endl;
    return 0;
}
```

上述代码细节很多，请读者仔细体会，并能独立写出来。

15 数码问题。八数码问题只有 9!种状态，对更大的问题，例如 4×4 棋盘的 15 数码问题，有 16! ≈ 2×10^{13} 种状态，如果仍然用数组存储状态，远远不够用。此时需要更好的算法[①]。

【习题】

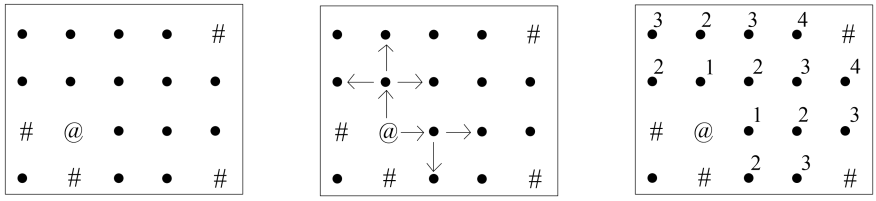
poj 1077 Eight，八数码问题。另外，在学过下节的 A*算法后，可重新做这道题。

4.3.3 BFS 与 A*算法

1. 用 BFS 求最短路

最短路是图论的一个基本问题，有很多复杂的算法。不过，在特殊的地图中，BFS 也是很好的最短路算法。下面仍然以 hdu 1312"Red and Black"的方格图为例，现在的任务是求两点之间的最短路。

在下图中，黑点'•'表示可以走的路，'#'表示不能走。求起点'@'到所有黑点'•'的最短距离。



(a)迷宫图　　(b)BFS 过程　　(c)最短距离

图 4.4 用 BFS 找最短路

方法很简单，从'@'出发，用 BFS 搜索所有点，记录到达每个点时经过的步数，即可得到从'@'到所有黑点的最短距离，图(c)标出了结果。

在这个例子中，BFS 搜最短路径的计算复杂度是 O(V+E)，非常好。

这个例子很特殊，图是方格形的，相邻两点之间的距离相同，也就是说，绕路肯定更远；BFS 先扩展到的路径，距离肯定是最短的。

如果相邻点的距离不同，绕路可能更近，BFS 就不适用了。关于最短路的通用算法，请阅读"10.9 最短路"的内容。

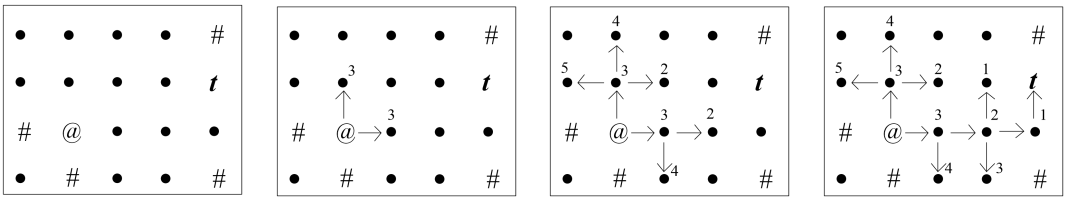
① 八数码的多种解法，例如双向广搜、A*、IDA*等，参考 https://www.cnblogs.com/zufezzt/p/5659276.html（永久网址：perma.cc/YV2V-GT6C ）

下面的 A*算法是 BFS 的优化。

2. A*算法与最短路

BFS 是一种“盲目的”搜索技术，它在搜索的过程中，并不理会目标在哪里，只顾自己乱走，当然，最后总会到达终点。

稍微改变 hdu 1312 的方格图，见下图(a)，现在的任务是求起点'@'到终点't'的最短路。



(a)起点和终点　(b)第 1 轮搜索　(c)第 2 轮搜索　(d)最短距离

图 4.5 启发式搜索

如果仍然用 BFS 求解，程序会搜索所有的点，直到遇到 *t* 点。不过，如果让一个人走这个图，他会一眼看出向右上方走，可以更快地找到到达 *t* 的最短路。人有“智能”，那么能否把这种智能教给程序呢？这就是“启发式”搜索算法。启发式搜索算法有很多种，A*算法是其中比较简单的一种。

简单地说，A*算法是“BFS+贪心”[①]。有关贪心法的解释，请阅读本书第 6 章。

在图 4.5(a)中，程序如何知道向右前方走，能更快到达 *t*？这里引入“曼哈顿距离”的概念。曼哈顿距离是指两个点在标准坐标系上的实际距离，在图中，就是@的坐标和 *t* 的坐标在横向和纵向的距离之和，它也被形象地称为“出租车距离”。

图 4.5(b)是从起点开始的第 1 轮 BFS 搜索，邻居点上标注的数字 3，是这个点到终点 *t* 的曼哈顿距离。图 4.5 (c)是第 2 轮搜索，标注 2 的点，是离终点更近的点，从这些点继续搜索；标注 4 和 5 的点距离终点远，先暂时停止搜索。经过多轮搜索，最后，图 4.5 (d)到达了终点 *t*。

在这个过程中，图中很多“不好的”点并不需要搜索到，从而优化了搜索过程。

上面的图例比较简单，如果起点和终点之间有很多障碍，搜索范围也会沿着障碍兜圈子，才能到达终点，不过，仍然有很多点不需要搜索。以下面的图为例，A 是起点，B 是终点，黑色方块是障碍。浅色阴影方块，是用曼哈顿距离进行启发式搜索所经过的部分；其它无色方块，是不需要搜索的。搜索结束后，得到一条最短路，见图中虚线。

① 这个网页用动画演示了 BFS、A*、Dijkstra 算法的原理，并给出了比较详细的伪代码描述，非常值得一看：https://www.redblobgames.com/pathfinding/a-star/introduction.html（永久网址：https://perma.cc/N2DB-5LDY）

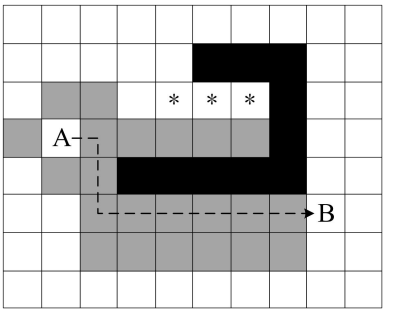


图 4.6　A*搜索求最短路

这个方法就是 A*算法，下面给出它的一般性描述。

在搜索过程中，用一个评估函数对当前情况进行评估，得到最好的状态，从这个状态继续搜索，直到目标。设 *x* 是当前所在的状态，*f(x)*是对 *x* 的评估函数，有：

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

*g(x)*表示从初始态到 x 的实际代价，它不体现 *x* 和终点的关系。

*h(x)*表示 *x* 到终点的最优路径的评估，它就是“启发式”信息，把 *h(x)*称为启发函数。很显然，*h(x)*决定了 A*算法的优劣。

特别需要注意的是，*h(x)*不能漏掉最优解。

在上面的例子中，曼哈顿距离就是启发函数 *h(x)*。曼哈顿距离是一种简单而且常用的启发函数。

在上面这个例子中，可以看出，A*算法包含了 BFS 和贪心算法：

（1）如果 $h(x) = 0$，有 $f(x) = g(x)$，就是普通的 BFS 算法；

（2）如果 $g(x) = 0$，有 $f(x) = h(x)$，就是贪心算法，此时图中标注’*’的方块也会被访问到。

3. A*算法与八数码问题

八数码问题也可以用 A*算法进行优化。考虑三种估价函数：

（1）以不在目标位置的数码的个数为估价函数；

（2）以不在目标位置的数码与目标位置的曼哈顿距离为估价函数；

（3）以逆序数[①]作为估价函数。

第（2）种比第（1）种好，可作为八数码问题的估价函数。

4.3.4 双向广搜

双向广搜是 BFS 的增强版。

前面提到，可以把 BFS 想象成在一个平静的池塘丢一颗石头，激起的波浪一层层扩散到整个空间，直到到达目标，就得到了从起点到目标点的最优路径。那么，如果同时在起点和

① 逆序数可以用来判断八数码是否有解。

目标点向对方做 BFS，两个石头激起的波浪，向对方扩散，将在中间某个位置遇到，此时即得到了最优路径。绝大多数情况下，双向广搜比只做一次 BFS，搜索的空间要减少很多，从而更有效率。

从上面的描述可知，双向广搜的应用场合是：知道起点和终点，并且正向和逆向都能进行搜索。

下面是一个典型的双向广搜问题。

hdu 1401 Solitaire
有一个 8x8 的棋盘，上面有 4 颗棋子，棋子可以上下左右移动。给定一个初始状态，和一个目标状态，问能否在 8 步内到达。

题目确定了起点和终点，十分适合双向 BFS。要求 8 步之内到达，可以从起点和终点分别开始，各自广搜 4 步，如果出现交点则说明可达。读者可以练习此题，程序比较繁琐，有很多细节需要处理，但是难度不高。

前一节讲解的八数码问题，也非常适合使用双向广搜技术进行优化。

【习题】

hdu 1401；

hdu 3567 “Eight II”，用双向广搜解决八数码问题。

4.4 DFS

4.4.1 DFS 和递归

hdu 1312 题有另外一种解决方案，即 4.3.1 节中提到的“一只老鼠走迷宫”。设 num 是到达的砖块数量，算法过程描述如下：

（1）在初始位置，令 num=1，标记这个位置已经走过；

（2）左、上、右、下 4 个方向，按顺时针顺序选一个能走的方向，走一步；

（3）在新的位置，num++，标记这个位置已经走过；

（4）继续前进，如果无路可走，回退到上一步，换个方向再走；

（5）继续以上过程，直到结束。

在以上过程中，能够访问到所有合法的砖块，并且每个砖块只访问一次，不会重复访问（回退不算重复）。

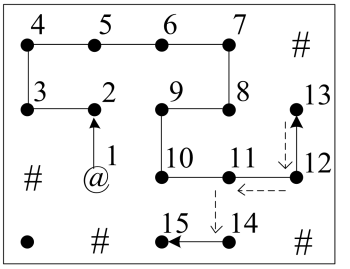


图 4.7　DFS 过程

hdu 1312 的路线如下：从 1 到 13，能一直走下去。在 13 这个位置，到底了不能再走，按顺序回退到 12、11；在 11 这个位置，换个方向又能走到 14、15。到达 15 后，发现不能再走下去，那么再按顺序倒退：14 → 11 → 10 → 9 → 8 → 7 → 6 → 5 → 4 → 3 → 2 → 1，在这个过程中，发现全部都没有新路，最后退回到起点，结束。

为加深对递归的理解，这里再次给出递归返回的完整顺序：13 → 12 → 15 → 14 → 11 → 10 → 9 → 8 → 7 → 6 → 5 → 4 → 3 → 2 → 1。

在这个过程中，最重要的特点是：在一个位置，只要有路，就一直走到最深处，直到无路可走，再退回一步，看在上一步的位置能不能换个方向继续往下走。这样就遍历了所有可能走到的位置。

这个思路就是深度搜索。从初始状态出发，下一步可能有多种状态；选其中一个状态深入，到达新的状态；直到无法继续深入，就回退到前一步，转移到其它状态，然后再深入下去。最后，遍历完所有可以到达的状态，并得到最终的解。

上述过程用 DFS 实现是最简单的，代码比 BFS 短很多。

下面是代码。读者可以在 DFS()函数中，打印走过的位置，以及回退的情况。从打印的信息能看出，到达 15 后，程序确实是逐步回退到起点的。

```
//用 DFS()替换 5.3.1 节程序中的 BFS()，并在 main()中的相同位置调用它
void DFS(int dx, int dy){
    room[dx][dy] = '#';                    //标记这个位置，表示已经走过
     // cout<<"walk:"<<dx<<dy<<endl; //在此处打印走过的位置，验证是否符合
    num++;
    for(int i = 0; i < 4; i++) {       //左、上、右、下，4 个方向顺时针深搜
       int newx = dx + dir[i][0];
       int newy = dy + dir[i][1];
       if(CHECK(newx, newy) && room[newx][newy] == '.'){
            DFS(newx, newy);
            // cout<<"    back:"<<dx<<dy<<endl;
            //在此处打印回退的点的坐标，观察深搜到底后，回退的情况
```

```
                //例如到达最后的 15 这个位置后，会一直退到起点
                //即打印出 14-11-10-9-8-7-6-5-4-3-2-1。这也是递归程序返回的过程
            }
        }
    }
```

4.4.2 回溯与剪枝

前面提到的 DFS 搜索，基本的操作是：将所有子结点全部扩展出来，再选取最新的一个结点进行扩展。

不过，很多情况下，用递归列举出所有的路径，可能会因为数量太大而超时。由于很多子结点是不符合条件的，可以在递归的时候，“看到不对头就撤退”，中途停止扩展并返回。这个思路就是回溯，回溯中用于减少子结点扩展的函数是剪枝函数。

大部分 DFS 搜索题目，都需要用到回溯的思路。其难度主要在于扩展子结点的时候，如何构造停止递归并返回的条件。这需要通过大量练习有关题目，才能熟练应用。

八皇后问题是经典的回溯与剪枝的应用。

八皇后问题。在棋盘上放置 8 个皇后，使得它们不同行、不同列、不同对角线。N 皇后问题是八皇后问题的扩展。

如果用暴力方法，先排列出所有的棋局，然后一一判断、去除非法的棋局，请读者自己思考复杂度有多大。

下面以四皇后问题为例描述解题过程。图 4.8 中，从第一行开始放皇后：第一行从左到右有 4 种方案，产生 4 个子结点；第二行，排除同列和斜线，扩展新的子结点，注意不用排除同行，因为第二行和第一行已经不同行；继续扩展第三和第四行，结束。

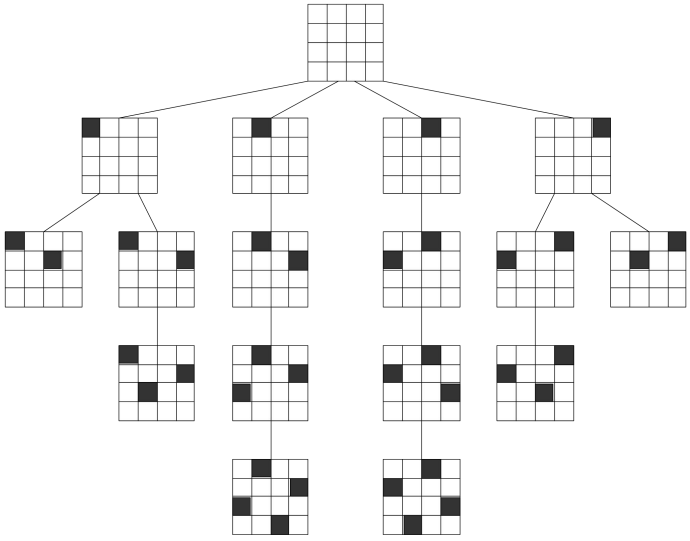


图 4.8 四皇后问题的搜索树

上图用 BFS 和 DFS 都能实现。前文说过，DFS 代码比 BFS 简洁很多。下面用 DFS 解决。

关键问题：在扩展结点时，如何去掉不符合条件的子结点？

设左上角是原点(0,0)，已经放好的皇后的坐标是(i, j)，不同行、不同列、不同斜线的新皇后的坐标是(r, c)，它们的关系是：

(1)横向，不同行：$i \neq r$。

(2)纵向，不同列：$j \neq c$。

(3)斜对角：从(i, j)向斜对角走 a 步，那么新坐标(r, c)有以下 4 种情况：左上角(i-a, j-a)、右上(i+a, j-a)、左下(i-a, j+a)、右下(i+a, j+a)，综合起来就是$|i - r| = |j - c|$。新皇后的位置不能放在斜线上，需满足$|i - r| \neq |j - c|$。

下面是 hdu 2553 的代码，求解 N 皇后问题，N <= 10。

hdu 2553　N 皇后问题

```
#include<bits/stdc++.h>
using namespace std;
int n, tot = 0;
int col[12] = {0};
bool check(int c, int r) {              //检查是否和已经放好的皇后冲突
     for(int i = 0; i < r; i++)
        if(col[i] == c || (abs(col[i]-c) == abs(i -r))) //取绝对值
            return false;
     return true;
}
void DFS(int r) {                    //一行一行地放皇后，这一次是第 r 行
     if(r == n) {              //所有皇后都放好了，递归返回
       tot++;                   //统计合法的棋局个数
       return;
     }
     for(int c = 0; c < n; c++)       //在每一列放皇后
        if(check(c, r)){               //检查是否合法
            col[r] = c;                //在第 r 行的 c 列放皇后
            DFS(r+1);                  //继续放下一行皇后
        }
}
int main() {
    int ans[12]={0};
```

```
    for(n = 0; n <= 10; n++){       //算出所有 n 皇后的答案。先打表不然会超时
        memset(col,0,sizeof(col)); //清空，准备计算下一个 N 皇后问题
        tot = 0;
        DFS(0);
        ans[n] = tot;                //打表
    }
    while(cin >> n) {
        if(n==0)
           return 0;
        cout << ans[n] << endl;
    }
    return 0;
}
```

N 皇后问题的 DFS 回溯程序非常简单，关键有两处：一是如何递归，二是如何剪枝和回溯。上述程序中有很多细节：

（1）打表。在 main()中，提前算出了从 1 到 10 所有的 N 皇后问题的答案，并存储在数组中，等读取输入后立刻输出。如果不打表，而是等输入 n 后再单独计算输出，会超时。

（2）递归搜索 DFS()。递归程序十分简洁：把第一个皇后按行放到棋盘上，然后递归放置其它的皇后，直到放完。

（3）回溯判断 check()。判断新放置的皇后和已经放好的皇后在横向、纵向、斜对角方向是否冲突。其中横向并不需要判断，因为递归的时候，已经是按不同的行放置的。

（4）模块化编程。例如 check()的内容很少，其实可以直接写在 DFS()内部，不用单独写成一个函数。但是单独写成函数，把功能模块化，好处很多：逻辑清晰、容易查错等等。建议写程序的时候，尽量把能分开的的功能单独写成函数，可以大大减少编码和调试的时间。

（5）复杂度。上述程序中，DFS()一行行地放皇后，复杂度 O(n!)；check()检查冲突，复杂度 O(n)；总复杂度是 O(n×n!)。N = 10 时，已经到千万数量级。读者可以自己在程序中统计运行次数。经本书作者验证，N = 11 时，计算了 900 万次，N = 12，计算 5 千万次。因此，对于 N > 11 的 N 皇后问题，需要用新的方法[①]。

【习题】

poj 2531 Network Saboteur；

poj1416 Shredding Company；

poj2676 Sudoku；

---

① 用数据结构舞蹈链(Dancing Links)，或者位运算可以较快地解决 N=15 的 N 皇后问题。更大的 N，例如当 N=27 时，有 2.34×10^{17} 个解。N 皇后问题是一个 NP 完全问题，不存在多项式时间的算法。

poj1129 Channel Allocation；

hdu 1175 连连看；

hdu 5113 Black And White。

4.4.3 迭代加深搜索

有这样一些题目，它们的搜索树很特别：不仅很深，而且很宽；深度可能到无穷，宽度也可能极广。如果直接用 DFS，会陷入递归无法返回；如果直接用 BFS，队列空间会爆炸。

此时可以采用一种结合了 DFS 和 BFS 思想的搜索方法，即迭代加深搜索（IDDFS，Iterative deepening DFS）。具体的操作方法是：

（1）先设定搜索深度为 1，用 DFS 搜索到第 1 层即停止。也就是说，用 DFS 搜索一个深度为 1 的搜索树；

（2）如果没有找到答案，再设定深度为 2，用 DFS 搜索前 2 层即停止。也就是说，用 DFS 搜索一个深度为 2 的搜索树；

（3）继续设定深度为 3、4……逐步扩大 DFS 的搜索深度，直到找到答案。

这个迭代过程，在每一层的广度上，采用了 BFS 搜索的思想，在具体编程实现上则是 DFS 的。

一个经典的例子是“埃及分数”。

埃及分数[①]

在古埃及，人们使用单位分数的和（形如 1/a 的，a 是自然数）表示一切有理数。如：2/3=1/2+1/6，但不允许 2/3=1/3+1/3，因为加数中有相同的。对于一个分数 a/b，表示方法有很多种，但是哪种最好呢？首先，加数少的比加数多的好，其次，加数个数相同的，最小的分数越大越好。如：

19/45=1/3 + 1/12 + 1/180，

19/45=1/3 + 1/15 + 1/45

19/45=1/3 + 1/18 + 1/30,

19/45=1/4 + 1/6 + 1/180

19/45=1/5 + 1/6 + 1/18

最好的是最后一种，因为 1/18 比 1/180、1/45、1/30、1/180 都大。 给出 a, b (0 < a < b < 1000)，编程计算最好的表达方式。

这一题显然是搜索，可以按图 4.9 建立搜索树。每一层的元素是分子为 1、分母递增的分数；从上往下的一个分支，就是一个这个分支上所有的分数相加的组合；找到合适的组合就退出。解答树的规模很大，深度可能无限，每一层的宽度也可能无限。

① https://loj.ac/problem/10022

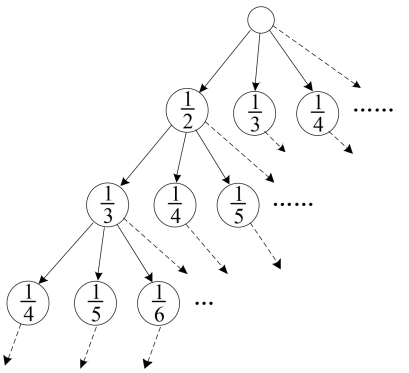


图 4.9 深度和宽度极大的搜索树

这种情况下适合使用迭代加深搜索。过程是：

（1）DFS 到第 1 层，只包括一个分数，如果满足要求就退出；

（2）DFS 前 2 层，是 2 个分数的和，例如 1/2 + 1/3、1/2 + 1/4、1/2 + 1/5、…、1/3 + 1/4、…等等，找到合适的答案就退出；

（3）DFS 前 3 层…

按上述步骤，能搜索到所有可能的组合，并且规避了直接使用 DFS 或 BFS 的弊端。

4.4.4 IDA*

IDA*是对迭代加深搜索 IDDFS 的优化。可以把 IDA*看成 A*算法思想在迭代加深搜索中的应用。

IDDFS 仍然是一种“盲目”的搜索方法，只是把搜索范围约束到了可行的空间内。如果在进行 IDDFS 的时候，能预测出当前 DFS 的状态，不再继续深入下去，那么就可以直接返回，不再继续，从而提高了效率。

这个预测就是在 IDDFS 中增加一个估价函数。在某个状态，经过函数计算，发现后续搜索无解，就返回。简单地说，就是 IDDFS 的过程中，利用估价函数进行剪枝操作。

下面这个例题说明了 IDDFS 和估价函数之间的关系。

poj 3134　Power Calculus

给定数 x 和 n，求 x^n，只能用乘法和除法，算过的结果可以被利用。问最少算多少次就够了。n<=1000。

这一题等价于：从数字 1 开始，用加减法，最少算多少次能得到 n。

搜索的范围是：每一步搜索，用前一步得出的值和之前产生的所有值进行加、减运算得到新的值，判断这个值是否等于 n。

这一题的麻烦在于，每一步搜索，新值的数量增长极快。如果直接用 DFS，深度可能有 1000，可能会溢出；如果用 BFS，也可能超出队列范围。

这一题用 IDDFS 非常合适，再用估价函数进行剪枝，可以高效地完成计算。

（1）IDDFS：指定递归深度，每一次做 DFS 时不超过这个深度；

（2）估价函数：如果当前的值用最快的方式（连续乘 2，倍增）都不能到达 n，停止用这个值继续 DFS。

poj 3134 代码

```
#include <iostream>
using namespace std;
int val[1010];                              //存一个搜索路径上每一步的计算结果
int pos, n;
bool ida(int now, int depth){
     if(now > depth)  return false;     //IDDFS：大于当前设定的 DFS 深度，退出
     if(val[pos] << (depth - now) < n)
         return false;                 //估价函数：用最快的倍增，都不能到达 n，退出
     if(val[pos] == n)  return true;    //当前结果等于 n，搜索结束
     pos ++ ;
     for(int i = 0 ; i < pos ; i ++) {
        val[pos] = val[pos-1] + val[i]; //上一个数与前面所有的数相加
        if(ida(now + 1, depth))   return true;
        val[pos] = abs(val[pos-1] - val[i]);//上一个数与前面所有的数相减
        if(ida(now + 1, depth))   return true;
     }
     pos -- ;
     return false;
}
int main(){
    while(cin>>n && n){
        int depth;
        for(depth = 0 ; ; depth ++){ //每次只 DFS 到深度 depth
            val[pos = 0] = 1;          //初始值是 1
            if(ida(0, depth)) break; //每次都从 0 层开始 DFS 到第 depth 层
        }
        cout << depth << endl;
    }
    return 0;
}
```

【习题】

hdu 1560 DNA sequence，经典 IDA*题目；

hdu 1667 The Rotation Game，经典 IDA*题目。

4.5 小结

DFS 和 BFS 是算法设计中的基本技术，是基础的基础。

这两种算法都能遍历搜索树的所有结点，区别在于如何扩展下一个结点。DFS 扩展子结点的子结点，搜索路径越来越深，适合采用栈这种数据结构，并用递归算法来实现；BFS 扩展子结点的兄弟结点，搜索路径越来越宽，适合用队列来实现。

1. 复杂度

DFS 和 BFS 对所有的点和边做了一次遍历，即对每个结点均做一次且仅做一次访问。设点的数量是 V，连接点的边总数是 E，那么总复杂度是 O(V+E)，看起来复杂度并不高。但是，有的问题的 V 和 E 本身就是指数级的，例如八数码问题的状态，是 O(n!)的。因此，在搜索时，需要用到剪枝、回溯、双向广搜、迭代加深、A*、IDA*等方法，尽量减少搜索的范围，使访问的总次数远远小于 O(V+E)。

2. 应用场合

DFS 一般用递归实现，代码比 BFS 更短。如果题目能用 DFS 解决，可以优先使用它。

当然，一些问题更适合用 DFS，另一些问题更适合用 BFS。一般情况下，BFS 是求解最优解的较好的方法，例如像迷宫这样的求最短路径问题，应该用 BFS，具体内容见第 10 章图论中的“最短路”；而 DFS 多用于求可行解。在“第 10 章 图论”中，还有大量应用 BFS 和 DFS 的例子。