第 8 章　回溯法

1. 在 $n \times n$ 格的棋盘上放置彼此不受攻击的 n 个皇后。按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。用非递归算法解决该问题。下图是一个 8 个皇后的例子，8 个皇后彼此不受攻击。

C/C++代码：

```
#include <stdio.h>
#include <stdlib.h>

void output(int n, int* a){
    for(int i=1; i<=n; i++){
        for(int j=1; j<=n; j++){
            if(j == a[i])
                printf("Q  ");
            else
                printf("*  ");
        }
        printf("\n");
    }
}
int check(int k, int* a) {
    for(int i=1; i<=k-1; i++)
        if( (abs(a[i]-a[k])==abs(i-k)) ||(a[i]==a[k]) )
            return 0;
    return 1;
}
int main(){
    int k=1, n, sum=0;
    scanf("%d", &n);
    int a[n+1];
    for(int i=1; i<=n; i++)
        a[i] =0;
    while(k>0) {
        a[k] = a[k] +1;
        while((a[k]<=n) && (check(k, a)==0) )  //为第 k 个皇后搜索位置
            a[k] = a[k] +1;
        if(a[k]<=n){
```

```
            if(k==n)
                { output(n, a); printf("\n"); sum = sum + 1;   } //找到一种放置
            else{
                k = k +1;   //继续为第 k+1 个皇后找位置
                a[k] =0;     //下一个皇后要从头开始搜索
            }
        }else
            k = k -1;              //回溯
    }
    printf("% d 皇后问题共有% d 种摆放方案", n, sum);
    return 0;
}
```

Python 代码：

```
def output(n, a):
    # 打印当前棋盘布局
    for i in range(1, n + 1):
        for j in range(1, n + 1):
            if j == a[i]:
                print("Q  ", end="")
            else:
                print("*   ", end="")
        print()

def check(k, a):
    # 检查第 k 个皇后放置在 a[k] 列是否和其他皇后冲突
    for i in range(1, k):
        if abs(a[i] - a[k]) == abs(i - k) or a[i] == a[k]:
            return False
    return True

def n_queen(n):
    a = [0] * (n + 1)  # 存储每一行皇后的位置
    k =1  # 从第 1 行开始
    sum_solutions =0  # 解的数量

    while k > 0:
        a[k] +=1  # 尝试将第 k 个皇后放置在下一个位置
        while a[k] <= n and not check(k, a):
            a[k] +=1  # 如果当前位置无效，继续尝试下一个位置
```

```
        if a[k] <= n:
            if k == n:    # 如果已经放置了 n 个皇后，输出当前解
                output(n, a)
                print()  # 输出换行
                sum_solutions +=1  # 找到一个解
            else:
                k +=1  # 向下放置下一个皇后
                a[k] =0  # 下一个皇后从第一列开始
        else:
            k -=1  # 如果没有有效的位置，回溯到上一行，重新尝试

    print(f"{n} 皇后问题共有 {sum_solutions} 种摆放方案")

if __name__ == "__main__":
    n =int(input())
    n_queen(n)
```

2. 在 $n \times n$ 格的棋盘上放置彼此不受攻击的 n 个皇后。按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。用递归算法解决该问题。

C/C++代码：

```
#include <stdio.h>
int n, sum=0, a[101], c[201], d[201];

void output(){
    for(int i=1; i<=n; i++){
        for(int j=1; j<=n; j++){
            if(j == a[i])
                printf("Q  ");
            else
                printf("*  ");
        }
        printf("\n");
    }
    sum = sum +1;
}
void swap(int t1, int t2) {
    int t;
    t=a[t1];
    a[t1]= a[t2];
```

```
    a[t2]=t;
}

void try1(int t) {
    int j;
    if (t>n)
      { output(); printf("\n"); }
    else
        for(j=t; j<=n; j++) {
            swap(t, j);
            if (c[t+a[t]]==0 && d[t-a[t]+n]==0) {
                c[t+a[t]]=1;    d[t-a[t]+n]=1;
                try1(t+1);
                c[t+a[t]]=0;    d[t-a[t]+n]=0;
            }
            swap(t, j);
        }
}
int main(){
    int i;
    scanf("%d", &n);

    for(int i=1; i<=n; i++)
        a[i] = i;
    for(int i=1; i<=n; i++){
        c[i]=0;
        c[n+i]=0;
        d[i]=0;
        d[n+i]=0;
    }
    try1(1);
    printf("%d皇后问题共有%d种摆放方案", n, sum);
    return 0;
}
```

Python 代码：

```
def output(n, a):
    # 打印当前棋盘布局
    for i in range(1, n + 1):
        for j in range(1, n + 1):
```

```
            if j == a[i]:
                print("Q  ", end="")
            else:
                print("*  ", end="")
        print()

def swap(a, t1, t2):
    # 交换 a[t1] 和 a[t2]
    a[t1], a[t2] = a[t2], a[t1]

def try1(n, t, a, c, d):
    # 回溯方法
    global sum
    if t > n:
        output(n, a)
        print()  # 输出换行
        sum += 1
    else:
        for j in range(t, n + 1):
            swap(a, t, j)
            if c[t + a[t]] == 0 and d[t - a[t] + n] == 0:
                c[t + a[t]] =1
                d[t - a[t] + n] =1
                try1(n, t +1, a, c, d)
                c[t + a[t]] =0
                d[t - a[t] + n] =0
            swap(a, t, j)

def n_queen(n):
    a = [0] * (n + 1)  # 存储每一行皇后的位置
    c = [0] * (2 * n + 1)  # 用于检测列的使用情况
    d = [0] * (2 * n + 1)  # 用于检测两个对角线的使用情况
    for i in range(1, n + 1):
        a[i] = i# 初始化位置
    try1(n, 1, a, c, d)
    print(f"{n} 皇后问题共有 {sum} 种摆放方案")

if __name__ == "__main__":
    n =int(input())
```

```
    sum = 0  # 解的数量
    n_queen(n)
```

3. 把从 1 到 20 这 20 个数摆成一个环，要求相邻的两个数的和是一个素数。

C/C++代码：

```
#include <stdio.h>
#include <stdbool.h>
#include <math.h>

#define MAX_N20 // 假设 n 的最大值为 20

int n; // 输入的 n
int ring[MAX_N]; // 存储当前的环
bool used[MAX_N +1]; // 记录数字是否被使用过
int count =0; // 记录满足条件的环的数量

//检查一个数是否为素数
bool isPrime(int num) {
    if (num < 2) return false;
    for (int i = 2; i <= sqrt(num); i++) {
        if (num %  i == 0) return false;
    }
    return true;
}

//回溯搜索所有满足条件的环
void backtrack(int pos) {
    // 如果已经填满环，检查首尾和是否为素数
    if (pos == n) {
        if (isPrime(ring[0] + ring[n - 1])) {
            // 输出当前环
            for (int i = 0; i < n; i++) {
                printf("% d ", ring[i]);
            }
            printf(" \n");
            count++; // 计数器加 1
        }
        return;
    }
```

```
    // 尝试将每个未使用的数字放入当前位置
    for (int i = 1; i <= n; i++) {
        if (!used[i]) {
            // 检查当前数字与前一个数字的和是否为素数
            if (pos == 0 || isPrime(ring[pos - 1] + i)) {
                ring[pos] = i; // 放入当前数字
                used[i] = true; // 标记为已使用
                backtrack(pos +1); // 递归填下一个位置
                used[i] = false; // 回溯，撤销选择
            }
        }
    }
}

int main() {
    // 输入 n
    scanf("% d", &n);
    // 初始化 used 数组
    for (int i = 1; i <= n; i++) {
        used[i] = false;
    }
    // 开始回溯搜索
    backtrack(0);
    // 输出满足条件的环的总数
    printf("% d 大小的环一共有% d 个素数环 \n", n, count);
    return 0;
}
```

Python 代码：

```
import sys
import math

MAX_N =20  # 假设 n 的最大值为 20

#检查一个数是否为素数
def is_prime(num):
    if num < 2:
        return False
    for i in range(2, int(math.sqrt(num)) + 1):
        if num %  i == 0:
```

```
            return False
    return True

#回溯搜索所有满足条件的环
def backtrack(pos, n, ring, used, count):
    # 如果已经填满环，检查首尾和是否为素数
    if pos == n:
        if is_prime(ring[0] + ring[-1]):
            # 输出当前环
            print(" ".join(map(str, ring)))
            count[0] += 1  # 计数器加 1
        return

    # 尝试将每个未使用的数字放入当前位置
    for i in range(1, n + 1):
        if not used[i]:
            # 检查当前数字与前一个数字的和是否为素数
            if pos == 0 or is_prime(ring[pos - 1] + i):
                ring[pos] = i# 放入当前数字
                used[i] =True  # 标记为已使用
                backtrack(pos +1, n, ring, used, count)  # 递归填下一个位置
                used[i] =False  # 回溯，撤销选择

def main():
    # 输入 n
    n =int(sys.stdin.readline())
    # 初始化 used 数组
    used = [False] * (n + 1)
    # 初始化 ring 数组
    ring = [0] * n
    # 初始化计数器
    count = [0]
    # 开始回溯搜索
    backtrack(0, n, ring, used, count)
    # 输出满足条件的环的总数
    print(f"{n}大小的环一共有{count[0]}个素数环")

if __name__ == "__main__":
    main()
```

4. 找 n 个数中 r 个数的组合。

C/C++代码：

```
#include <stdio.h>

int n, r;
int combination[20]; // 存储当前组合

//回溯函数生成组合
void backtrack(int pos, int start) {
    // 已选够 r 个数，输出组合
    if (pos == r) {
        for (int i = 0; i < r; i++) {
            printf("%d ", combination[i]);
        }
        printf("\n");
        return;
    }

    // 从 start 开始递减选择数字
    for (int i = start; i >= 1; i--) {
        // 剪枝：剩余数字不足以填满剩余位置
        if (i < r - pos) break;

        combination[pos] = i; // 选择当前数字
        backtrack(pos +1, i - 1); // 递归选择下一个更小的数字
    }
}

int main() {
    scanf("%d %d", &n, &r);
    backtrack(0, n); // 从位置 0 开始，最大可选数字为 n
    return 0;
}
```

Python 代码：

```
def backtrack(pos, start, n, r, combination):
    # 已选够 r 个数，输出组合
    if pos == r:
        print(" ".join(map(str, combination[: r])))
        return
```

```
    # 从 start 开始递减选择数字
    for i in range(start, 0, -1):
        # 剪枝: 剩余数字不足以填满剩余位置
        if i < r - pos:
            break

        combination[pos] = i# 选择当前数字
        backtrack(pos +1, i - 1, n, r, combination)  # 递归选择下一个更小的数字

def main():
    n, r =map(int, input().split())
    combination = [0] *  r  # 初始化组合数组
    backtrack(0, n, n, r, combination)

if __name__ == "__main__":
    main()
```

5. 有一个 $n \times m$ 的棋盘，在某个点(x, y)上有一个马，要求计算出马到达棋盘上任意一点最少要走几步。

C/C++代码：

```
#include <stdio.h>
#include <string.h>

//马的 8 个可能移动方向
int dx[] = {-2, -2, -1, -1, 1, 1, 2, 2};
int dy[] = {-1, 1, -2, 2, -2, 2, -1, 1};

int n, m;
int visited[20][20];   // 标记已访问的位置
int steps[20][20];     // 记录到达每个位置的最少步数

//检查位置是否合法
int isValid(int x, int y) {
    return x >= 0 && x < n && y >= 0 && y < m;
}

//回溯函数
void backtrack(int x, int y, int step) {
    // 如果当前位置的步数已经比之前记录的步数大，则剪枝
    if (visited[x][y] && step >= steps[x][y]) {
```

```
        return;
    }

    // 更新当前点的最少步数
    steps[x][y] = step;
    visited[x][y] =1;

    // 尝试所有可能的方向
    for (int i = 0; i < 8; i++) {
        int nx = x + dx[i];
        int ny = y + dy[i];

        if (isValid(nx, ny)) {
            backtrack(nx, ny, step +1);
        }
    }
}

int main() {
    int x, y;
    scanf("% d % d % d % d", &n, &m, &x, &y);
    // 初始化
    memset(visited, 0, sizeof(visited));
    memset(steps, -1, sizeof(steps));   // -1 表示不可达
    // 从起点开始回溯(注意输入的坐标从 1 开始, 数组从 0 开始)
    backtrack(x-1, y-1, 0);
    // 输出结果
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            printf("% d", steps[i][j]);
            if (j < m-1) printf(" ");
        }
        printf("\n");
    }
    return 0;
}
```

Python 代码：

```
#马的 8 个可能移动方向
dx = [-2, -2, -1, -1, 1, 1, 2, 2]
```

```
dy = [-1, 1, -2, 2, -2, 2, -1, 1]

def is_valid(x, y, n, m):
    """检查位置是否合法"""
    return 0 <= x < n and 0 <= y < m

def backtrack(x, y, step, n, m, visited, steps):
    """回溯函数"""
    # 如果当前位置的步数已经比之前记录的步数大，则剪枝
    if visited[x][y] and step >= steps[x][y]:
        return

    # 更新当前点的最少步数
    steps[x][y] = step
    visited[x][y] =1

    # 尝试所有可能的方向
    for i in range(8):
        nx = x + dx[i]
        ny = y + dy[i]

        if is_valid(nx, ny, n, m):
            backtrack(nx, ny, step +1, n, m, visited, steps)

def main():
    # 输入棋盘大小和起始位置
    n, m, x, y =map(int, input().split())

    # 初始化
    visited = [[0] *  m for _ in range(n)]  # 标记已访问的位置
    steps = [[-1] *  m for _ in range(n)]   # -1 表示不可达

    # 从起点开始回溯(注意输入的坐标从 1 开始，数组从 0 开始)
    backtrack(x -1, y - 1, 0, n, m, visited, steps)

    # 输出结果
    for i in range(n):
        row = []
        for j in range(m):
```

```
            row.append(str(steps[i][j]))
        print(" ".join(row))

if __name__ == "__main__":
    main()
```

6. 给定无向连通图 G 和 m 种不同的颜色，用这些颜色为图 G 的各顶点着色，每个顶点着一种颜色。如果有一种着色法使 G 中每条边的 2 个顶点着不同颜色，则称这个图是 m 可着色的。图的 m 着色问题是给定图 G 和 m 种颜色，找出所有不同的着色法。

C/C++代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//最大顶点数
#define MAXN20

int n, k, m;                          // n: 顶点数; k: 边数; m: 颜色数
int graph[MAXN][MAXN];                // 邻接矩阵表示图
int colors[MAXN];                     // 记录每个顶点的颜色
int solutions =0;                     // 记录不同着色方案数

//检查当前顶点着色是否合法
int isValid(int vertex, int color) {
    for (int i = 1; i <= n; i++) {
        if (graph[vertex][i] && colors[i] == color) {
            return 0;    // 相邻顶点颜色相同，不合法
        }
    }
    return 1;
}

//回溯法枚举所有着色方案
void colorGraph(int vertex) {
    if (vertex > n) {
        // 所有顶点已着色，记录方案数
        solutions++;
        return;
    }
```

```
    // 尝试为当前顶点选择颜色
    for (int color = 1; color <= m; color++) {
        if (isValid(vertex, color)) {
            colors[vertex] = color;   // 给当前顶点着色
            colorGraph(vertex +1);    // 递归着色下一个顶点
            colors[vertex] =0;        // 回溯，撤销当前顶点的颜色
        }
    }
}

int main() {
    // 输入顶点数、边数和颜色数
    //printf("请输入顶点数、边数和颜色数: ");
    scanf("% d % d % d", &n, &k, &m);
    // 初始化图的邻接矩阵
    memset(graph, 0, sizeof(graph));
    // 输入边
    //printf("请输入每条边的信息(格式: u v): \n");
    for (int i = 0; i < k; i++) {
        int u, v;
        scanf("% d % d", &u, &v);
        graph[u][v] =1;
        graph[v][u] =1;   // 无向图对称
    }
    // 回溯计算不同的着色方案
    colorGraph(1);
    // 输出结果
    printf("% d\n", solutions);
    return 0;
}
```

Python 代码：

```
#最大顶点数
MAXN =20

#图的邻接矩阵
graph = [[0] * (MAXN + 1) for _ in range(MAXN + 1)]
#记录每个顶点的颜色
colors = [0] * (MAXN + 1)
#记录不同着色方案数
```

```
solutions =0

#检查当前顶点着色是否合法
def is_valid(vertex, color, n):
    for i in range(1, n + 1):
        if graph[vertex][i] and colors[i] == color:
            return False  # 相邻顶点颜色相同，不合法
    return True
#回溯法枚举所有着色方案
def color_graph(vertex, n, m):
    global solutions
    if vertex > n:
        # 所有顶点已着色，记录方案数
        solutions +=1
        return
    # 尝试为当前顶点选择颜色
    for color in range(1, m + 1):
        if is_valid(vertex, color, n):
            colors[vertex] = color# 给当前顶点着色
            color_graph(vertex +1, n, m)  # 递归着色下一个顶点
            colors[vertex] =0  # 回溯，撤销当前顶点的颜色

def main():
    global solutions
    # 输入顶点数、边数和颜色数
    n, k, m =map(int, input().split())
    # 初始化图的邻接矩阵
    for _ in range(k):
        u, v =map(int, input().split())
        graph[u][v] =1
        graph[v][u] =1  # 无向图对称
    # 回溯计算不同的着色方案
    color_graph(1, n, m)

    # 输出结果
    print(f"{solutions}")

if __name__ == "__main__":
    main()
```