手写代码必备手册(C++版本)

作者：[@soulmachine](http://www.yanjiuyanjiu.com/)

1. 暴力枚举法
   1. 算法思想

生成 -- 测试法。

* 1. 简单枚举
     1. 分数拆分

输入正整数k，找到所有的正整数 ，使得 。

样例输入：

2

12

样例输出：

2

1/2=1/6+1/3

1/2=1/4+1/4

8

1/12=1/156+1/13

1/12=1/84+1/14

1/12=1/60+1/15

1/12=1/48+1/16

1/12=1/36+1/18

1/12=1/30+1/20

1/12=1/28+1/21

1/12=1/24+1/24

【**分析**】

既然说找出所有的x,y，枚举对象自然就是他们了。可问题在于：枚举范围如何？从1/12=1/156+1/13，可以看出，x可以比y大很多。难道要无休止地枚举下去？当然不是。由于，有，因此 ，即。这样，只需要在2k范围之类枚举y，然后根据y算出x即可。

* 1. 枚举排列
     1. 生成1~n的排列
     2. 生成可重复集合的排列
     3. 下一个排列
  2. 子集生成
     1. 增量构造法
     2. 位向量法
     3. 二进制法

1. 回溯法
   1. 算法思想

当把问题分成若干步骤并递归求解时，如果当前步骤没有合法选择，则函数将返回上一级递归调用，这种现象称为回溯。正是因为这个原因，枚举递归算法常被称为回溯法。

回溯法的几个组成部分：递归，深度遍历，剪枝。

回溯法比暴力枚举法快的原因，在于：暴力枚举法，是每生成一个完整的解答后，再来判断这个解答是否合法，而回溯法则在生成每一步中都进行判断，而不是等一个答案生成完毕后再来判断，这样，在每一步进行剪枝，减少了大量的废答案。

* 1. 八皇后问题

在8×8的棋盘上，放置8个皇后，使得她们互不攻击，每个皇后的攻击范围是同行、同列和同对角线，要求找出所有解。如图2-1所示。

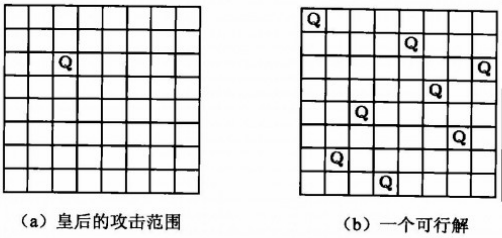


图2-1 八皇后问题

【分析】

最简单的暴力枚举方法是，从64个格子中选一个子集，使得子集含有8个格子，且任意两个格子都不在同一行、同一列或同一个对角线上。这正是子集枚举问题，然而64个格子的子集有264个，太大了，这并不是一个很好的模型。

第二个思路是，从64个格子中选8个格子，这是组合生成问题。根据组合数学，有 种方案，比第一种方案优秀，但仍然不够好。

经过思考不难发现，由于每一行只能放一个皇后，那么第一行有8种选择，第二行有7中选择，…，第8行有1中选择，总共有8!=40320个方案。如果用C[x]表示第x行皇后的列编号，则问题变成了一个全排列生成问题，枚举量不会超过8!。

【代码】

#include <stdio.h>

#include <stdlib.h>

#define QUEENS 8 // 皇后的个数，也是棋盘的长和宽

static int total = 0; // 可行解的总数

static int C[QUEENS] = {0}; // C[i]表示第i行皇后所在的列编号

/\*\*

\* @brief 输出所有皇后的坐标.

\* @return 无

\*/

void output() {

int i;

for(i = 0; i < QUEENS; ++i) {

printf("(%d,%d) ", i, C[i]);

}

printf("\n");

}

/\*\*

\* @brief 检查当前位置(current, column)能否放置皇后.

\*

\* @param[in] current 当前行

\* @return 能则返回1，不能则返回0

\*/

static int check(const int current, const int column) {

int ok = 1;

int j;

for(j = 0; j < current; ++j) {

// 两个点的坐标为(current, column), (j, C[j])

// 检查是否在同一列，或对角线上

if(column == C[j] || current - j == column - C[j] || current - j == C[j] - column) {

ok = 0;

break;

}

}

return ok;

}

/\*\*

\* @brief 八皇后，回溯法

\*

\* @param[in] current 搜索当前行，该再哪一列上放一个皇后

\* @return 无

\*/

static void search(const int current) {

if(current == QUEENS) { // 递归边界，只要走到了这里，意味着找到了一个可行解

++total;

// output();

} else {

int i;

for(i = 0; i < QUEENS; ++i) { // 一列一列的试

const int ok = check(current, i);

if(ok) { // 如果合法，继续递归

C[current] = i;

search(current + 1);

}

}

}

}

// 表示已经放置的皇后

// 占据了哪些列

static int columns[QUEENS] = {0};

// 占据了哪些主对角线

static int principal\_diagonals[2 \* QUEENS] = {0};

// 占据了哪些副对角线

static int counter\_diagonals[2 \* QUEENS] = {0};

/\*\*

\* @brief 检查当前位置(current, column)能否放置皇后.

\*

\* @param[in] current 当前行

\* @return 能则返回1，不能则返回0

\*/

static int check2(const int current, const int column) {

return columns[column] == 0 && principal\_diagonals[current + column] == 0 && counter\_diagonals[current - column + QUEENS] == 0;

}

/\*\*

\* @brief 八皇后，回溯法，更优化的版本，用空间换时间

\*

\* @param[in] current 搜索当前行，该再哪一列上放一个皇后

\* @return 无

\*/

static void search2(const int current) {

if(current == QUEENS) { // 递归边界，只要走到了这里，意味着找到了一个可行解

++total;

// output();

} else {

int i;

for(i = 0; i < QUEENS; ++i) { // 一列一列的试

const int ok = check2(current, i);

if(ok) { // 如果合法，继续递归

C[current] = i;

columns[i] = 1;

principal\_diagonals[current + i] = 1;

counter\_diagonals[current- i + QUEENS] = 1;

search(current + 1);

// 恢复环境

columns[i] = 0;

principal\_diagonals[current + i] = 0;

counter\_diagonals[current- i + QUEENS] = 0;

}

}

}

}

int main(int argc, char\* args[]) {

total = 0;

search(0);

printf("%d\n", total);

total = 0;

search2(0);

printf("%d\n", total);

return 0;

}