

## 第 13 章 回溯



### Abstract

我们如同迷宫中的探索者，在前进的道路上可能会遇到困难。

回溯的力量让我们能够重新开始，不断尝试，最终找到通往光明的出口。

## 13.1 回溯算法

回溯算法 (backtracking algorithm) 是一种通过穷举来解决问题的方法，它的核心思想是从一个初始状态出发，暴力搜索所有可能的解决方案，当遇到正确的解则将其记录，直到找到解或者尝试了所有可能的选择都无法找到解为止。

回溯算法通常采用“深度优先搜索”来遍历解空间。在“二叉树”章节中，我们提到前序、中序和后序遍历都属于深度优先搜索。接下来，我们利用前序遍历构造一个回溯问题，逐步了解回溯算法的工作原理。

### 例题一

给定一棵二叉树，搜索并记录所有值为 7 的节点，请返回节点列表。

对于此题，我们前序遍历这棵树，并判断当前节点的值是否为 7，若是，则将该节点的值加入结果列表 `res` 之中。相关过程实现如图 13-1 和以下代码所示：

```
// === File: preorder_traversal_i_compact.c ===
```

```
/* 前序遍历：例题一 */
```

```
void preOrder(TreeNode *root) {  
    if (root == NULL) {  
        return;  
    }  
    if (root->val == 7) {  
        // 记录解  
        res[resSize++] = root;  
    }  
    preOrder(root->left);  
    preOrder(root->right);  
}
```

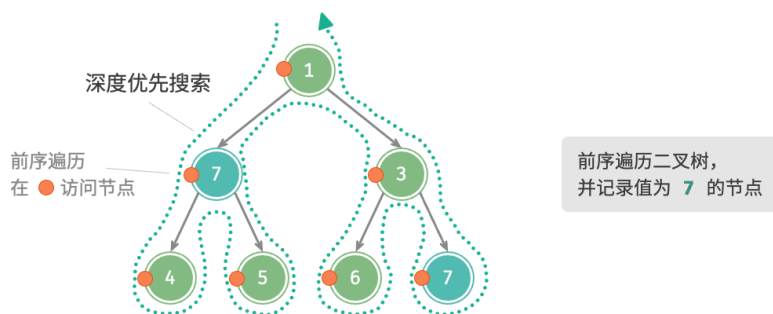


图 13-1 在前序遍历中搜索节点

### 13.1.1 尝试与回退

之所以称之为回溯算法，是因为该算法在搜索解空间时会采用“尝试”与“回退”的策略。当算法在搜索过程中遇到某个状态无法继续前进或无法得到满足条件的解时，它会撤销上一步的选择，退回到之前的状态，并尝试其他可能的选择。

对于例题一，访问每个节点都代表一次“尝试”，而越过叶节点或返回父节点的 `return` 则表示“回退”。

值得说明的是，回退并不仅仅包括函数返回。为解释这一点，我们对例题一稍作拓展。

#### 例题二

在二叉树中搜索所有值为 7 的节点，请返回根节点到这些节点的路径。

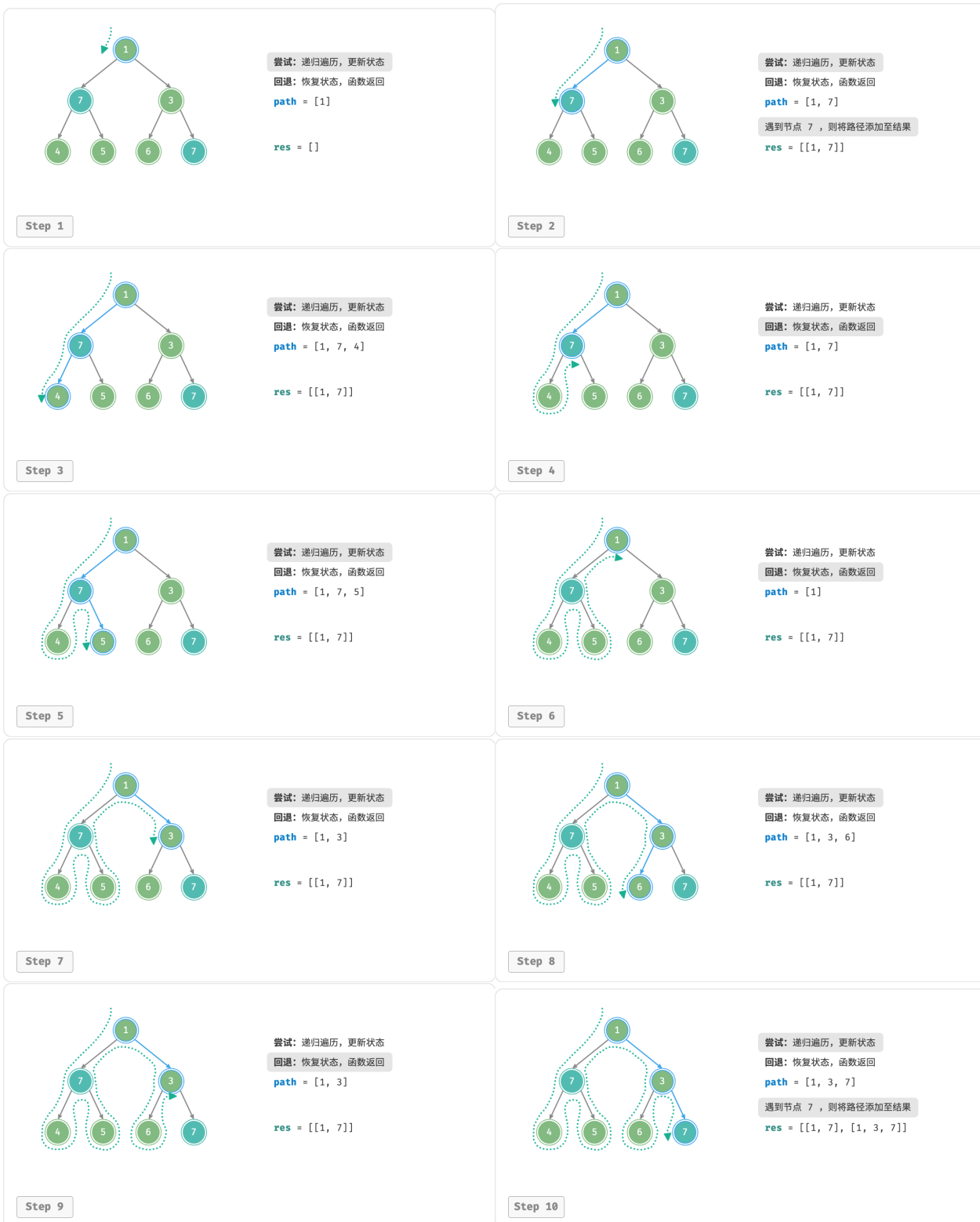
在例题一代码的基础上，我们需要借助一个列表 `path` 记录访问过的节点路径。当访问到值为 7 的节点时，则复制 `path` 并添加进结果列表 `res`。遍历完成后，`res` 中保存的就是所有的解。代码如下所示：

```
// === File: preorder_traversal_ii_compact.c ===

/* 前序遍历：例题二 */
void preOrder(TreeNode *root) {
    if (root == NULL) {
        return;
    }
    // 尝试
    path[pathSize++] = root;
    if (root->val == 7) {
        // 记录解
        for (int i = 0; i < pathSize; ++i) {
            res[resSize][i] = path[i];
        }
        resSize++;
    }
    preOrder(root->left);
    preOrder(root->right);
    // 回退
    pathSize--;
}
```

在每次“尝试”中，我们通过将当前节点添加进 `path` 来记录路径；而在“回退”前，我们需要将该节点从 `path` 中弹出，以恢复本次尝试之前的状态。

观察图 13-2 所示的过程，我们可以将尝试和回退理解为“前进”与“撤销”，两个操作互为逆向。



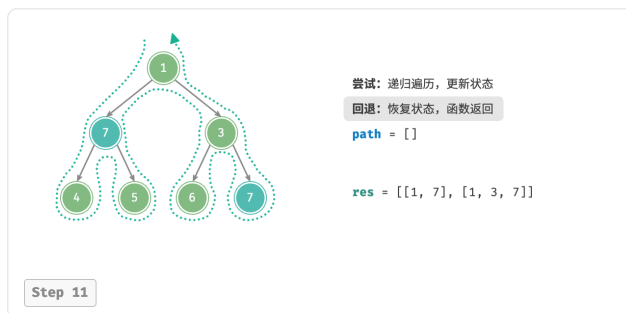


图 13-2 尝试与回退

### 13.1.2 剪枝

复杂的回溯问题通常包含一个或多个约束条件，约束条件通常可用于“剪枝”。

#### 例题三

在二叉树中搜索所有值为 7 的节点，请返回根节点到这些节点的路径，并要求路径中不包含值为 3 的节点。

为了满足以上约束条件，我们需要添加剪枝操作：在搜索过程中，若遇到值为 3 的节点，则提前返回，不再继续搜索。代码如下所示：

```
// === File: preorder_traversal_iii_compact.c ===
```

```
/* 前序遍历：例题三 */
```

```
void preOrder(TreeNode *root) {  
    // 剪枝  
    if (root == NULL || root->val == 3) {  
        return;  
    }  
    // 尝试  
    path[pathSize++] = root;  
    if (root->val == 7) {  
        // 记录解  
        for (int i = 0; i < pathSize; i++) {  
            res[resSize][i] = path[i];  
        }  
        resSize++;  
    }  
    preOrder(root->left);  
    preOrder(root->right);  
    // 回退  
    pathSize--;  
}
```

“剪枝”是一个非常形象的名词。如图 13-3 所示，在搜索过程中，我们“剪掉”了不满足约束条件的搜索分支，避免许多无意义的尝试，从而提高了搜索效率。



### 13.1.3 框架代码

接下来，我们尝试将回溯的“尝试、回退、剪枝”的主体框架提炼出来，提升代码的通用性。

在以下框架代码中，`state` 表示问题的当前状态，`choices` 表示当前状态下可以做出的选择：

```
/* 回溯算法框架 */
void backtrack(State *state, Choice *choices, int numChoices, State *res, int numRes) {
    // 判断是否为解
    if (isSolution(state)) {
        // 记录解
        recordSolution(state, res, numRes);
        // 不再继续搜索
        return;
    }
    // 遍历所有选择
    for (int i = 0; i < numChoices; i++) {
        // 剪枝：判断选择是否合法
        if (isValid(state, &choices[i])) {
            // 尝试：做出选择，更新状态
            makeChoice(state, &choices[i]);
            backtrack(state, choices, numChoices, res, numRes);
            // 回退：撤销选择，恢复到之前的状态
            undoChoice(state, &choices[i]);
        }
    }
}
```

接下来，我们基于框架代码来解决例题三。状态 `state` 为节点遍历路径，选择 `choices` 为当前节点的左子节点和右子节点，结果 `res` 是路径列表：

```
// === File: preorder_traversal_iii_template.c ===

/* 判断当前状态是否为解 */
bool isSolution(void) {
    return pathSize > 0 && path[pathSize - 1]->val == 7;
}

/* 记录解 */
void recordSolution(void) {
    for (int i = 0; i < pathSize; i++) {
        res[resSize][i] = path[i];
    }
    resSize++;
}

/* 判断在当前状态下，该选择是否合法 */
bool isValid(TreeNode *choice) {
    return choice != NULL && choice->val != 3;
}

/* 更新状态 */
void makeChoice(TreeNode *choice) {
    path[pathSize++] = choice;
}

/* 恢复状态 */
void undoChoice(void) {
    pathSize--;
}

/* 回溯算法：例题三 */
void backtrack(TreeNode *choices[2]) {
    // 检查是否为解
    if (isSolution()) {
        // 记录解
        recordSolution();
    }
    // 遍历所有选择
    for (int i = 0; i < 2; i++) {
        TreeNode *choice = choices[i];
        // 剪枝：检查选择是否合法
        if (isValid(choice)) {
            // 尝试：做出选择，更新状态
            makeChoice(choice);
            // 进行下一轮选择
            TreeNode *nextChoices[2] = {choice->left, choice->right};
            backtrack(nextChoices);
        }
    }
}
```



```
// 回退：撤销选择，恢复到之前的状态
undoChoice();
}
}
}
```

根据题意，我们在找到值为 7 的节点后应该继续搜索，因此需要将记录解之后的 `return` 语句删除。图 13-4 对比了保留或删除 `return` 语句的搜索过程。

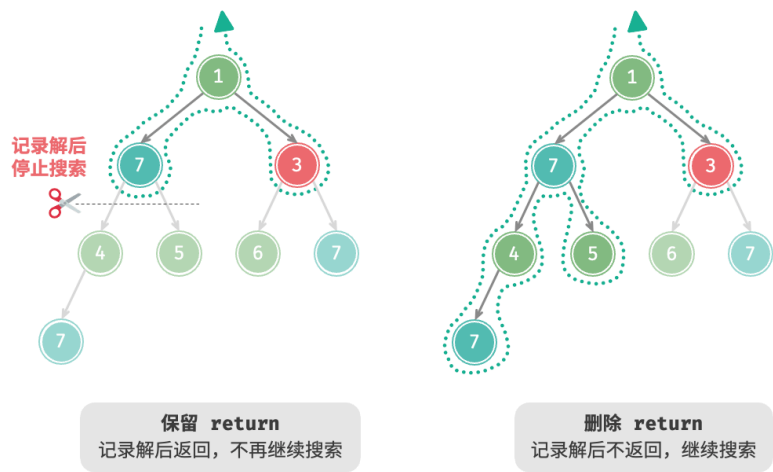


图 13-4 保留与删除 return 的搜索过程对比

相比基于前序遍历的代码实现，基于回溯算法框架的代码实现虽然显得啰唆，但通用性更好。实际上，许多回溯问题可以在该框架下解决。我们只需根据具体问题来定义 `state` 和 `choices`，并实现框架中的各个方法即可。

13.1.4 常用术语

为了更清晰地分析算法问题，我们总结一下回溯算法中常用术语的含义，并对照例题三给出对应示例，如表 13-1 所示。

表 13-1 常见的回溯算法术语

名词	定义	例题三
解 (solution)	解是满足问题特定条件的答案，可能有一个或多个	根节点到节点 7 的满足约束条件的所有路径
约束条件 (constraint)	约束条件是问题中限制解的可行性的条件，通常用于剪枝	路径中不包含节点 3



名词	定义	例题三
状态 (state)	状态表示问题在某一时刻的情况，包括已经做出的选择	当前已访问的节点路径，即 <code>path</code> 节点列表
尝试 (attempt)	尝试是根据可用选择来探索解空间的过程，包括做出选择，更新状态，检查是否为解	递归访问左（右）子节点，将节点添加进 <code>path</code> ，判断节点的值是否为 7
回退 (backtracking)	回退指遇到不满足约束条件的状态时，撤销前面做出的选择，回到上一个状态	当越过叶节点、结束节点访问、遇到值为 3 的节点时终止搜索，函数返回
剪枝 (pruning)	剪枝是根据问题特性和约束条件避免无意义的搜索路径的方法，可提高搜索效率	当遇到值为 3 的节点时，则不再继续搜索

**Tip**  
问题、解、状态等概念是通用的，在分治、回溯、动态规划、贪心等算法中都有涉及。

13.1.5 优点与局限性

回溯算法本质上是一种深度优先搜索算法，它尝试所有可能的解决方案直到找到满足条件的解。这种方法的优点在于能够找到所有可能的解决方案，而且在合理的剪枝操作下，具有很高的效率。

然而，在处理大规模或者复杂问题时，回溯算法的运行效率可能难以接受。

- **时间**：回溯算法通常需要遍历状态空间的所有可能，时间复杂度可以达到指数阶或阶乘阶。
- **空间**：在递归调用中需要保存当前的状态（例如路径、用于剪枝的辅助变量等），当深度很大时，空间需求可能会变得很大。

即便如此，回溯算法仍然是某些搜索问题和约束满足问题的最佳解决方案。对于这些问题，由于无法预测哪些选择可生成有效的解，因此我们必须对所有可能的选择进行遍历。在这种情况下，关键是如何优化效率，常见的效率优化方法有两种。

- **剪枝**：避免搜索那些肯定不会产生解的路径，从而节省时间和空间。
- **启发式搜索**：在搜索过程中引入一些策略或者估计值，从而优先搜索最有可能产生有效解的路径。

13.1.6 回溯典型例题

回溯算法可用于解决许多搜索问题、约束满足问题和组合优化问题。

**搜索问题**：这类问题的目标是找到满足特定条件的解决方案。

- **全排列问题**：给定一个集合，求出其所有可能的排列组合。
- **子集和问题**：给定一个集合和一个目标和，找到集合中所有和为目标和的子集。
- **汉诺塔问题**：给定三根柱子和一系列大小不同的圆盘，要求将所有圆盘从一根柱子移动到另一根柱子，每次只能移动一个圆盘，且不能将大圆盘放在小圆盘上。

**约束满足问题**：这类问题的目标是找到满足所有约束条件的解。

- $n$  皇后：在  $n \times n$  的棋盘上放置  $n$  个皇后，使得它们互不攻击。
- 数独：在  $9 \times 9$  的网格中填入数字  $1 \sim 9$ ，使得每行、每列和每个  $3 \times 3$  子网格中的数字不重复。
- 图着色问题：给定一个无向图，用最少的颜色给图的每个顶点着色，使得相邻顶点颜色不同。

**组合优化问题：**这类问题的目标是在一个组合空间中找到满足某些条件的最优解。

- 0-1 背包问题：给定一组物品和一个背包，每个物品有一定的价值和重量，要求在背包容量限制内，选择物品使得总价值最大。
- 旅行商问题：在一个图中，从一个点出发，访问所有其他点恰好一次后返回起点，求最短路径。
- 最大团问题：给定一个无向图，找到最大的完全子图，即子图中的任意两个顶点之间都有边相连。

请注意，对于许多组合优化问题，回溯不是最优解决方案。

- 0-1 背包问题通常使用动态规划解决，以达到更高的时间效率。
- 旅行商是一个著名的 NP-Hard 问题，常用解法有遗传算法和蚁群算法等。
- 最大团问题是图论中的一个经典问题，可用贪心算法等启发式算法来解决。

## 13.2 全排列问题

全排列问题是回溯算法的一个典型应用。它的定义是在给定一个集合（如一个数组或字符串）的情况下，找出其中元素的所有可能的排列。

表 13-2 列举了几个示例数据，包括输入数组和对应的所有排列。

表 13-2 全排列示例

输入数组	所有排列
[1]	[1]
[1, 2]	[1, 2], [2, 1]
[1, 2, 3]	[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]

### 13.2.1 无相等元素的情况

#### Question

输入一个整数数组，其中不包含重复元素，返回所有可能的排列。

从回溯算法的角度看，我们可以把生成排列的过程想象成一系列选择的结果。假设输入数组为 `[1, 2, 3]`，如果我们先选择 1，再选择 3，最后选择 2，则获得排列 `[1, 3, 2]`。回退表示撤销一个选择，之后继续尝试其他选择。

从回溯代码的角度看，候选集合 `choices` 是输入数组中的所有元素，状态 `state` 是直至今前已被选择的元素。请注意，每个元素只允许被选择一次，因此 `state` 中的所有元素都应该是唯一的。

如图 13-5 所示, 我们可以将搜索过程展开成一棵递归树, 树中的每个节点代表当前状态 `state`。从根节点开始, 经过三轮选择后到达叶节点, 每个叶节点都对应一个排列。

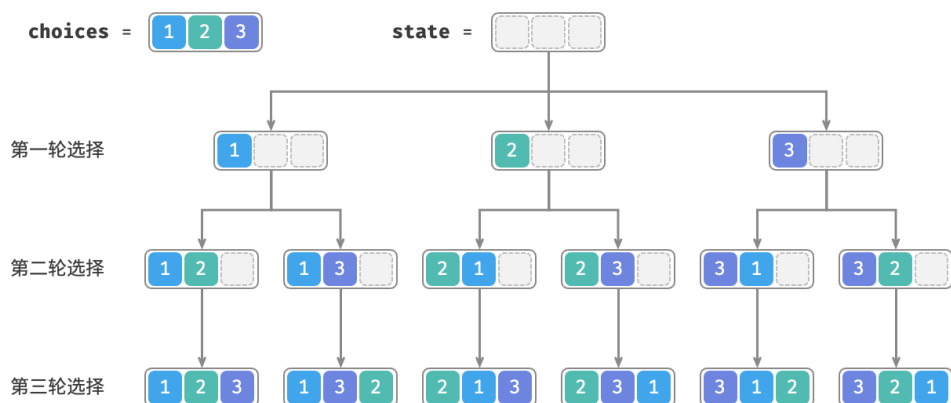


图 13-5 全排列的递归树

### 1. 重复选择剪枝

为了实现每个元素只被选择一次, 我们考虑引入一个布尔型数组 `selected`, 其中 `selected[i]` 表示 `choices[i]` 是否已被选择, 并基于它实现以下剪枝操作。

- 在做出选择 `choice[i]` 后, 我们就将 `selected[i]` 赋值为 `True`, 代表它已被选择。
- 遍历选择列表 `choices` 时, 跳过所有已被选择的节点, 即剪枝。

如图 13-6 所示, 假设我们第一轮选择 1, 第二轮选择 3, 第三轮选择 2, 则需要在第二轮剪掉元素 1 的分支, 在第三轮剪掉元素 1 和元素 3 的分支。

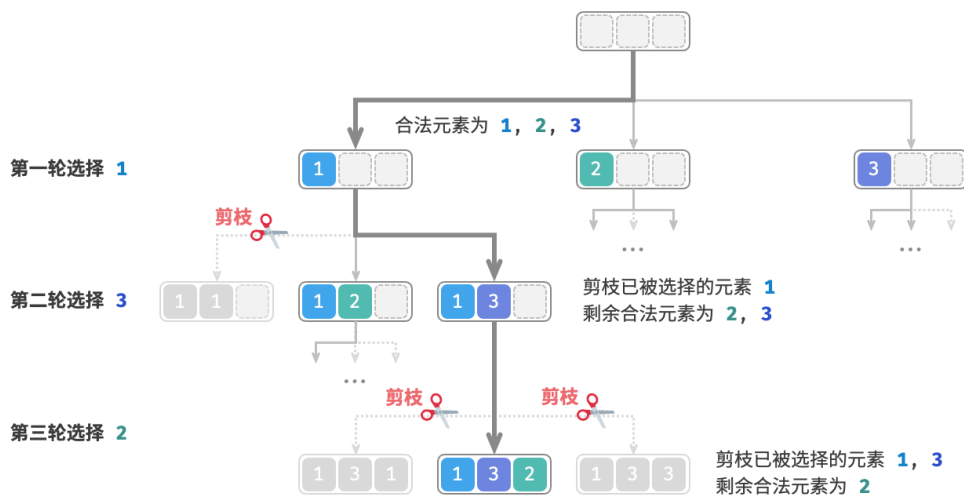


图 13-6 全排列剪枝示例

观察图 13-6 发现，该剪枝操作将搜索空间大小从  $O(n^n)$  减小至  $O(n!)$ 。

## 2. 代码实现

想清楚以上信息之后，我们就可以在框架代码中做“完形填空”了。为了缩短整体代码，我们不单独实现框架代码中的各个函数，而是将它们展开在 `backtrack()` 函数中：

```
// === File: permutations_i.c ===

/* 回溯算法：全排列 I */
void backtrack(int *state, int stateSize, int *choices, int choicesSize, bool *selected, int **res, int
↪ *resSize) {
    // 当状态长度等于元素数量时，记录解
    if (stateSize == choicesSize) {
        res[*resSize] = (int *)malloc(choicesSize * sizeof(int));
        for (int i = 0; i < choicesSize; i++) {
            res[*resSize][i] = state[i];
        }
        (*resSize)++;
        return;
    }
    // 遍历所有选择
    for (int i = 0; i < choicesSize; i++) {
        int choice = choices[i];
        // 剪枝：不允许重复选择元素
        if (!selected[i]) {
            // 尝试：做出选择，更新状态
            selected[i] = true;
            state[stateSize] = choice;
            // 进行下一轮选择
            backtrack(state, stateSize + 1, choices, choicesSize, selected, res, resSize);
            // 回退：撤销选择，恢复到之前的状态
            selected[i] = false;
        }
    }
}

/* 全排列 I */
int **permutationsI(int *nums, int numsSize, int *returnSize) {
    int *state = (int *)malloc(numsSize * sizeof(int));
    bool *selected = (bool *)malloc(numsSize * sizeof(bool));
    for (int i = 0; i < numsSize; i++) {
        selected[i] = false;
    }
    int **res = (int **)malloc(MAX_SIZE * sizeof(int *));
    *returnSize = 0;
```

```
backtrack(state, 0, nums, numsSize, selected, res, returnSize);

free(state);
free(selected);

return res;
}
```

### 13.2.2 考虑相等元素的情况

#### Question

输入一个整数数组，数组中可能包含重复元素，返回所有不重复的排列。

假设输入数组为  $[1, 1, 2]$ 。为了方便区分两个重复元素 1，我们将第二个 1 记为  $\hat{1}$ 。

如图 13-7 所示，上述方法生成的排列有一半是重复的。

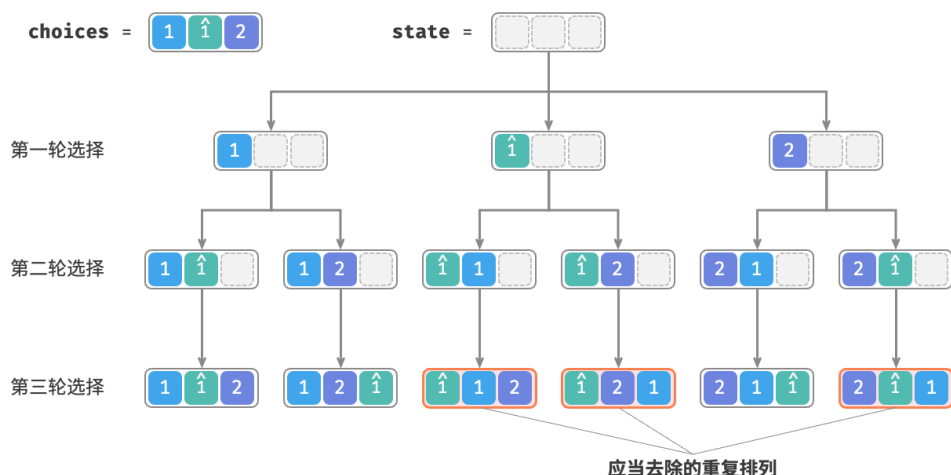


图 13-7 重复排列

那么如何去除重复的排列呢？最直接地，考虑借助一个哈希集合，直接对排列结果进行去重。然而这样做不够优雅，因为生成重复排列的搜索分支没有必要，应当提前识别并剪枝，这样可以进一步提升算法效率。

#### 1. 相等元素剪枝

观察图 13-8，在第一轮中，选择 1 或选择  $\hat{1}$  是等价的，在这两个选择之下生成的所有排列都是重复的。因此应该把  $\hat{1}$  剪枝。

同理，在第一轮选择 2 之后，第二轮选择中的 1 和  $\hat{1}$  也会产生重复分支，因此也应将第二轮的  $\hat{1}$  剪枝。

从本质上看，我们的目标是在某一轮选择中，保证多个相等的元素仅被选择一次。

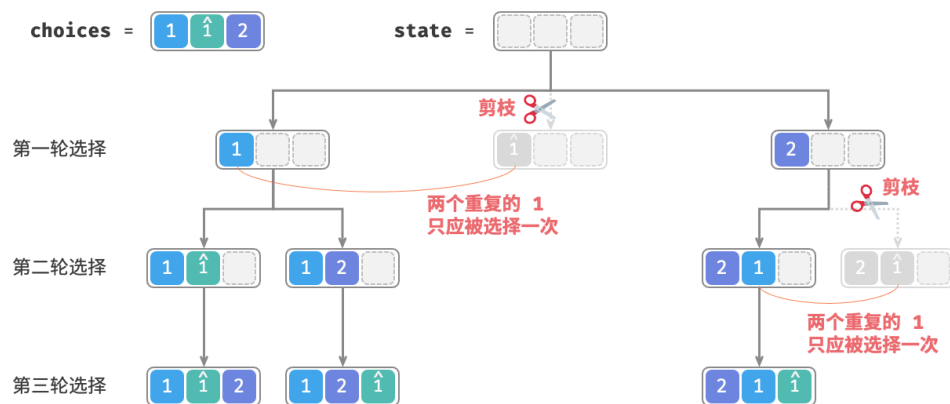


图 13-8 重复排列剪枝

## 2. 代码实现

在上一题的代码的基础上，我们考虑在每一轮选择中开启一个哈希集合 `duplicated`，用于记录该轮中已经尝试过的元素，并将重复元素剪枝：

```
// === File: permutations_ii.c ===

/* 回溯算法：全排列 II */
void backtrack(int *state, int stateSize, int *choices, int choicesSize, bool *selected, int **res, int
↪ *resSize) {
    // 当状态长度等于元素数量时，记录解
    if (stateSize == choicesSize) {
        res[*resSize] = (int *)malloc(choicesSize * sizeof(int));
        for (int i = 0; i < choicesSize; i++) {
            res[*resSize][i] = state[i];
        }
        (*resSize)++;
        return;
    }
    // 遍历所有选择
    bool duplicated[MAX_SIZE] = {false};
    for (int i = 0; i < choicesSize; i++) {
        int choice = choices[i];
        // 剪枝：不允许重复选择元素 且 不允许重复选择相等元素
        if (!selected[i] && !duplicated[choice]) {
            // 尝试：做出选择，更新状态
            duplicated[choice] = true; // 记录选择过的元素值
            selected[i] = true;
            state[stateSize] = choice;
            // 进行下一轮选择
```

```
        backtrack(state, stateSize + 1, choices, choicesSize, selected, res, resSize);
        // 回退：撤销选择，恢复到之前的状态
        selected[i] = false;
    }
}

/* 全排列 II */
int **permutationsII(int *nums, int numsSize, int *returnSize) {
    int *state = (int *)malloc(numsSize * sizeof(int));
    bool *selected = (bool *)malloc(numsSize * sizeof(bool));
    for (int i = 0; i < numsSize; i++) {
        selected[i] = false;
    }
    int **res = (int **)malloc(MAX_SIZE * sizeof(int *));
    *returnSize = 0;

    backtrack(state, 0, nums, numsSize, selected, res, returnSize);

    free(state);
    free(selected);

    return res;
}
```

假设元素两两之间互不相同，则  $n$  个元素共有  $n!$  种排列（阶乘）；在记录结果时，需要复制长度为  $n$  的列表，使用  $O(n)$  时间。因此时间复杂度为  $O(n!n)$ 。

最大递归深度为  $n$ ，使用  $O(n)$  栈帧空间。`selected` 使用  $O(n)$  空间。同一时刻最多共有  $n$  个 `duplicated`，使用  $O(n^2)$  空间。因此空间复杂度为  $O(n^2)$ 。

### 3. 两种剪枝对比

请注意，虽然 `selected` 和 `duplicated` 都用于剪枝，但两者的目标不同。

- **重复选择剪枝**：整个搜索过程中只有一个 `selected`。它记录的是当前状态中包含哪些元素，其作用是避免某个元素在 `state` 中重复出现。
- **相等元素剪枝**：每轮选择（每个调用的 `backtrack` 函数）都包含一个 `duplicated`。它记录的是在本轮遍历（`for` 循环）中哪些元素已被选择过，其作用是保证相等元素只被选择一次。

图 13-9 展示了两个剪枝条件的生效范围。注意，树中的每个节点代表一个选择，从根节点到叶节点的路径上的各个节点构成一个排列。



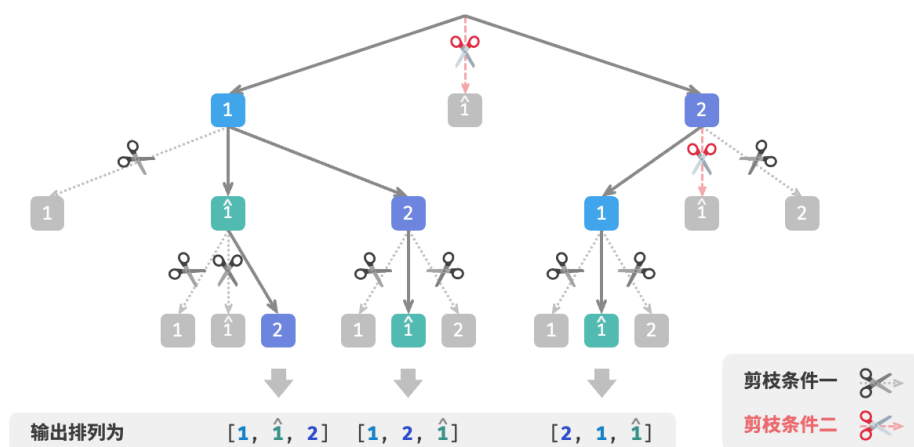


图 13-9 两种剪枝条件的作用范围

## 13.3 子集和问题

### 13.3.1 无重复元素的情况

#### Question

给定一个正整数数组 `nums` 和一个目标正整数 `target`，请找出所有可能的组合，使得组合中的元素和等于 `target`。给定数组无重复元素，每个元素可以被选取多次。请以列表形式返回这些组合，列表中不应包含重复组合。

例如，输入集合  $\{3, 4, 5\}$  和目标整数 9，解为  $\{3, 3, 3\}, \{4, 5\}$ 。需要注意以下两点。

- 输入集合中的元素可以被无限次重复选取。
- 子集不区分元素顺序，比如  $\{4, 5\}$  和  $\{5, 4\}$  是同一个子集。

#### 1. 参考全排列解法

类似于全排列问题，我们可以把子集的生成过程想象成一系列选择的结果，并在选择过程中实时更新“元素和”，当元素和等于 `target` 时，就将子集记录至结果列表。

而与全排列问题不同的是，本题集合中的元素可以被无限次选取，因此无须借助 `selected` 布尔列表来记录元素是否已被选择。我们可以对全排列代码进行小幅修改，初步得到解题代码：

```
// === File: subset_sum_i_naive.c ===

/* 回溯算法：子集和 I */
void backtrack(int target, int total, int *choices, int choicesSize) {
    // 子集和等于 target 时，记录解
```

```
    if (total == target) {
        for (int i = 0; i < stateSize; i++) {
            res[resSize][i] = state[i];
        }
        resColSizes[resSize++] = stateSize;
        return;
    }
    // 遍历所有选择
    for (int i = 0; i < choicesSize; i++) {
        // 剪枝: 若子集和超过 target, 则跳过该选择
        if (total + choices[i] > target) {
            continue;
        }
        // 尝试: 做出选择, 更新元素和 total
        state[stateSize++] = choices[i];
        // 进行下一轮选择
        backtrack(target, total + choices[i], choices, choicesSize);
        // 回退: 撤销选择, 恢复到之前的状态
        stateSize--;
    }
}

/* 求解子集和 I (包含重复子集) */
void subsetSumINaive(int *nums, int numsSize, int target) {
    resSize = 0; // 初始化解的数量为 0
    backtrack(target, 0, nums, numsSize);
}
```

向以上代码输入数组  $[3, 4, 5]$  和目标元素 9, 输出结果为  $[3, 3, 3], [4, 5], [5, 4]$ 。虽然成功找出了所有和为 9 的子集, 但其中存在重复的子集  $[4, 5]$  和  $[5, 4]$ 。

这是因为搜索过程是区分选择顺序的, 然而子集不区分选择顺序。如图 13-10 所示, 先选 4 后选 5 与先选 5 后选 4 是不同的分支, 但对应同一个子集。

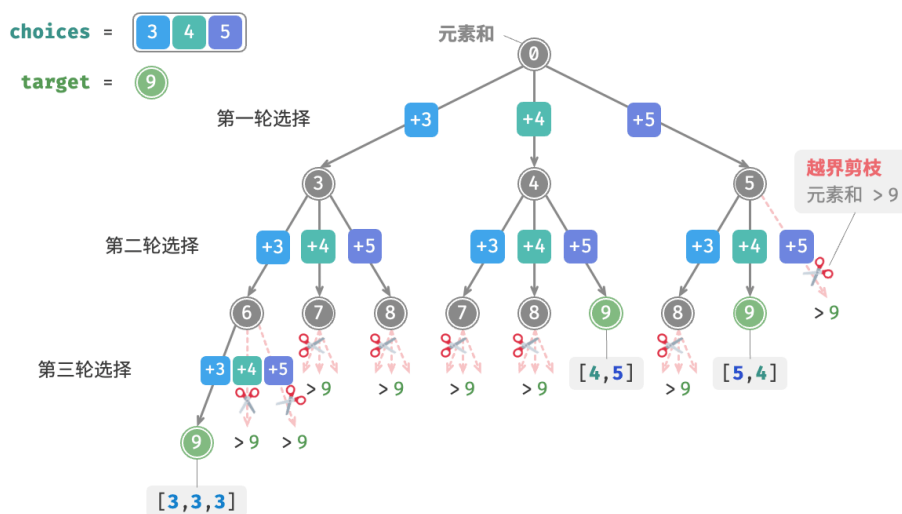


图 13-10 子集搜索与越界剪枝

为了去除重复子集，一种直接的思路是对结果列表进行去重。但这个方法效率很低，有两方面原因。

- 当数组元素较多，尤其是当 **target** 较大时，搜索过程会产生大量的重复子集。
- 比较子集（数组）的异同非常耗时，需要先排序数组，再比较数组中每个元素的异同。

## 2. 重复子集剪枝

我们考虑在搜索过程中通过剪枝进行去重。观察图 13-11，重复子集是在以不同顺序选择数组元素时产生的，例如以下情况。

1. 当第一轮和第二轮分别选择 3 和 4 时，会生成包含这两个元素的所有子集，记为  $[3, 4, \dots]$ 。
2. 之后，当第一轮选择 4 时，则第二轮应该跳过 3，因为该选择产生的子集  $[4, 3, \dots]$  和第 1. 步中生成的子集完全重复。

在搜索过程中，每一层的选择都是从左到右被逐个尝试的，因此越靠右的分支被剪掉的越多。

1. 前两轮选择 3 和 5，生成子集  $[3, 5, \dots]$ 。
2. 前两轮选择 4 和 5，生成子集  $[4, 5, \dots]$ 。
3. 若第一轮选择 5，则第二轮应该跳过 3 和 4，因为子集  $[5, 3, \dots]$  和  $[5, 4, \dots]$  与第 1. 步和第 2. 步中描述的子集完全重复。

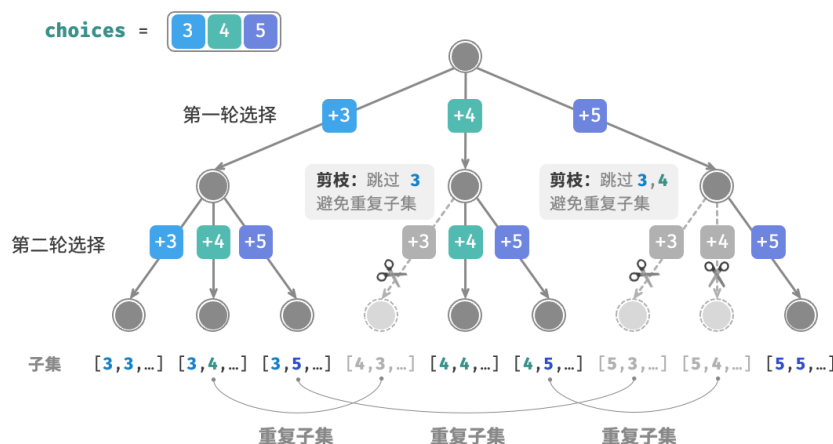


图 13-11 不同选择顺序导致的重复子集

总结来看，给定输入数组  $[x_1, x_2, \dots, x_n]$ ，设搜索过程中的选择序列为  $[x_{i_1}, x_{i_2}, \dots, x_{i_m}]$ ，则该选择序列需要满足  $i_1 \leq i_2 \leq \dots \leq i_m$ ，不满足该条件的选择序列都会造成重复，应当剪枝。

### 3. 代码实现

为实现该剪枝，我们初始化变量 `start`，用于指示遍历起始点。当做出选择  $x_i$  后，设定下一轮从索引  $i$  开始遍历。这样做就可以让选择序列满足  $i_1 \leq i_2 \leq \dots \leq i_m$ ，从而保证子集唯一。

除此之外，我们还对代码进行了以下两项优化。

- 在开启搜索前，先将数组 `nums` 排序。在遍历所有选择时，当子集和超过 `target` 时直接结束循环，因为后边的元素更大，其子集和一定超过 `target`。
- 省去元素和变量 `total`，通过在 `target` 上执行减法来统计元素和，当 `target` 等于 0 时记录解。

```
// === File: subset_sum_i.c ===

/* 回溯算法：子集和 I */
void backtrack(int target, int *choices, int choicesSize, int start) {
    // 子集和等于 target 时，记录解
    if (target == 0) {
        for (int i = 0; i < stateSize; ++i) {
            res[resSize][i] = state[i];
        }
        resColSizes[resSize++] = stateSize;
        return;
    }
    // 遍历所有选择
    // 剪枝二：从 start 开始遍历，避免生成重复子集
    for (int i = start; i < choicesSize; i++) {
```

```
// 剪枝一：若子集和超过 target，则直接结束循环
// 这是因为数组已排序，后边元素更大，子集和一定超过 target
if (target - choices[i] < 0) {
    break;
}
// 尝试：做出选择，更新 target, start
state[stateSize] = choices[i];
stateSize++;
// 进行下一轮选择
backtrack(target - choices[i], choices, choicesSize, i);
// 回退：撤销选择，恢复到之前的状态
stateSize--;
}
}

/* 求解子集和 I */
void subsetSumI(int *nums, int numsSize, int target) {
    qsort(nums, numsSize, sizeof(int), cmp); // 对 nums 进行排序
    int start = 0;                          // 遍历起始点
    backtrack(target, nums, numsSize, start);
}
```

图 13-12 所示为将数组 [3, 4, 5] 和目标元素 9 输入以上代码后的整体回溯过程。

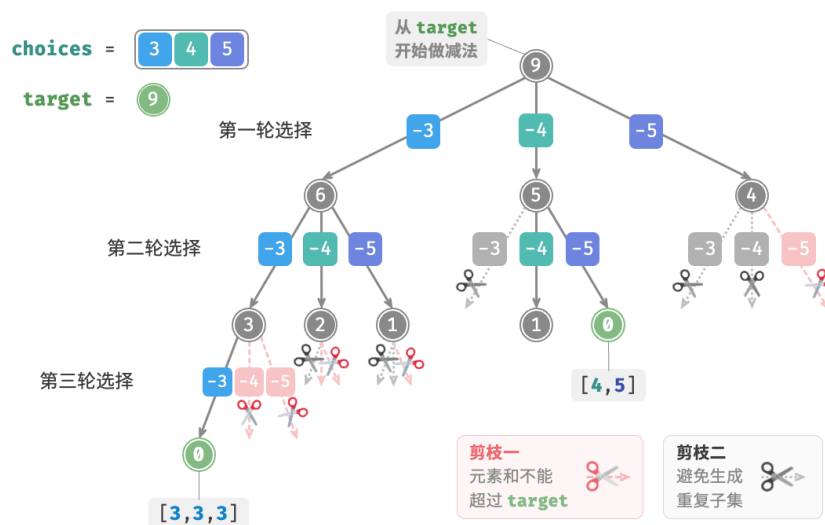


图 13-12 子集和 I 回溯过程

### 13.3.2 考虑重复元素的情况

#### Question

给定一个正整数数组 `nums` 和一个目标正整数 `target`，请找出所有可能的组合，使得组合中的元素和等于 `target`。给定数组可能包含重复元素，每个元素只可被选择一次。请以列表形式返回这些组合，列表中不应包含重复组合。

相比于上题，本题的输入数组可能包含重复元素，这引入了新的问题。例如，给定数组  $[4, \hat{4}, 5]$  和目标元素 9，则现有代码的输出结果为  $[4, 5], [\hat{4}, 5]$ ，出现了重复子集。

造成这种重复的原因是相等元素在某轮中被多次选择。在图 13-13 中，第一轮共有三个选择，其中两个都为 4，会产生两个重复的搜索分支，从而输出重复子集；同理，第二轮的两个 4 也会产生重复子集。

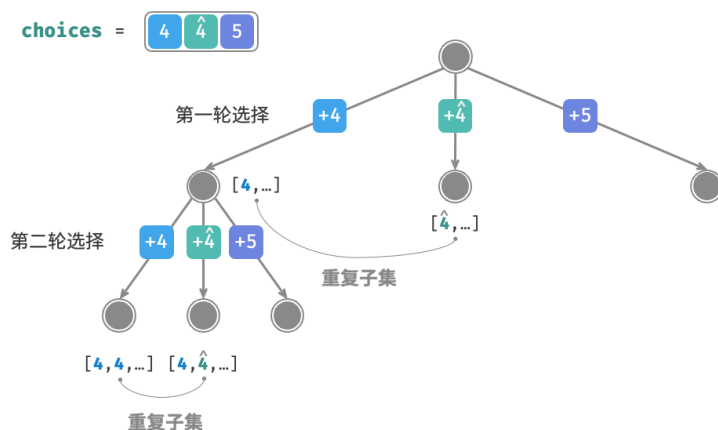


图 13-13 相等元素导致的重复子集

#### 1. 相等元素剪枝

为解决此问题，我们需要限制相等元素在每一轮中只能被选择一次。实现方式比较巧妙：由于数组是已排序的，因此相等元素都是相邻的。这意味着在某轮选择中，若当前元素与其左边元素相等，则说明它已经被选择过，因此直接跳过当前元素。

与此同时，本题规定每个数组元素只能被选择一次。幸运的是，我们也可以利用变量 `start` 来满足该约束：当做出选择  $x_i$  后，设定下一轮从索引  $i + 1$  开始向后遍历。这样既能去除重复子集，也能避免重复选择元素。

#### 2. 代码实现

```
// === File: subset_sum_ii.c ===
```

```
/* 回溯算法：子集和 II */
```

```
void backtrack(int target, int *choices, int choicesSize, int start) {
    // 子集和等于 target 时, 记录解
    if (target == 0) {
        for (int i = 0; i < stateSize; i++) {
            res[resSize][i] = state[i];
        }
        resColSizes[resSize++] = stateSize;
        return;
    }
    // 遍历所有选择
    // 剪枝二: 从 start 开始遍历, 避免生成重复子集
    // 剪枝三: 从 start 开始遍历, 避免重复选择同一元素
    for (int i = start; i < choicesSize; i++) {
        // 剪枝一: 若子集和超过 target, 则直接跳过
        if (target - choices[i] < 0) {
            continue;
        }
        // 剪枝四: 如果该元素与左边元素相等, 说明该搜索分支重复, 直接跳过
        if (i > start && choices[i] == choices[i - 1]) {
            continue;
        }
        // 尝试: 做出选择, 更新 target, start
        state[stateSize] = choices[i];
        stateSize++;
        // 进行下一轮选择
        backtrack(target - choices[i], choices, choicesSize, i + 1);
        // 回退: 撤销选择, 恢复到之前的状态
        stateSize--;
    }
}

/* 求解子集和 II */
void subsetSumII(int *nums, int numsSize, int target) {
    // 对 nums 进行排序
    qsort(nums, numsSize, sizeof(int), cmp);
    // 开始回溯
    backtrack(target, nums, numsSize, 0);
}
```

图 13-14 展示了数组 [4, 4, 5] 和目标元素 9 的回溯过程, 共包含四种剪枝操作。请你将图示与代码注释相结合, 理解整个搜索过程, 以及每种剪枝操作是如何工作的。



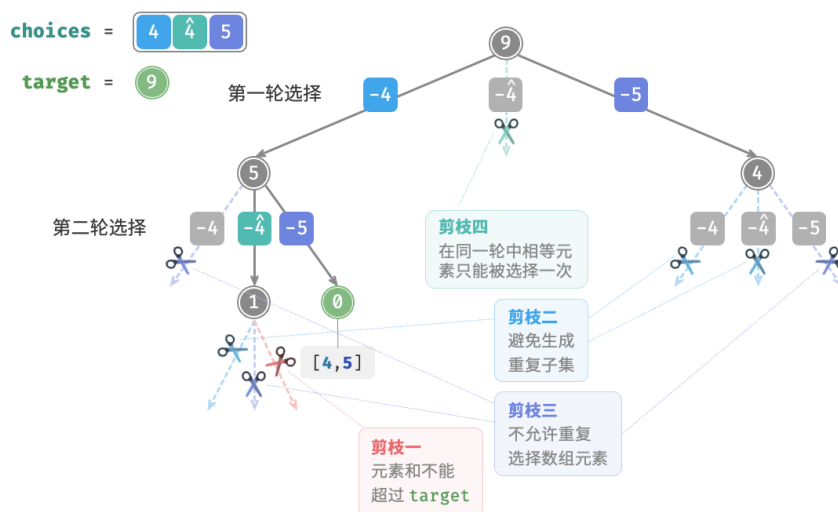


图 13-14 子集和 II 回溯过程

## 13.4 n 皇后问题

### Question

根据国际象棋的规则，皇后可以攻击与同处一行、一列或一条斜线上的棋子。给定  $n$  个皇后和一个  $n \times n$  大小的棋盘，寻找使得所有皇后之间无法相互攻击的摆放方案。

如图 13-15 所示，当  $n = 4$  时，共可以找到两个解。从回溯算法的角度看， $n \times n$  大小的棋盘共有  $n^2$  个格子，给出了所有的选择 choices。在逐个放置皇后的过程中，棋盘状态在不断地变化，每个时刻的棋盘就是状态 state。

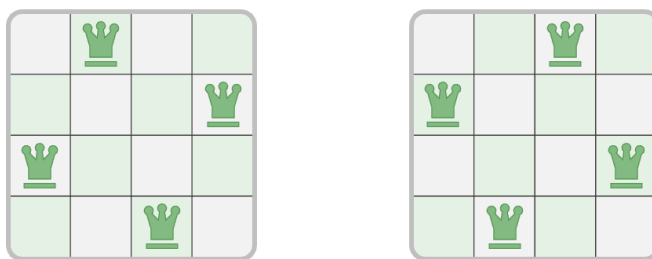


图 13-15 4 皇后问题的解

图 13-16 展示了本题的三个约束条件：多个皇后不能在同一行、同一列、同一条对角线上。值得注意的是，对角线分为主对角线 \ 和次对角线 / 两种。

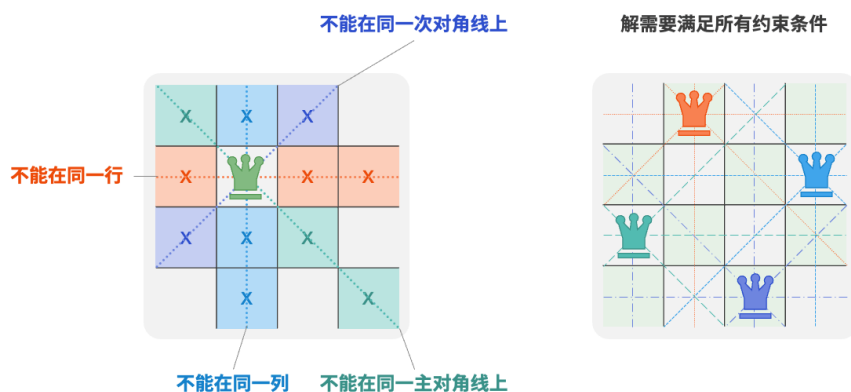


图 13-16 n 皇后问题的约束条件

### 1. 逐行放置策略

皇后的数量和棋盘的行数都为  $n$ ，因此我们容易得到一个推论：**棋盘每行都允许且只允许放置一个皇后。**

也就是说，我们可以采取逐行放置策略：从第一行开始，在每行放置一个皇后，直至最后一行结束。

图 13-17 所示为 4 皇后问题的逐行放置过程。受画幅限制，图 13-17 仅展开了第一行的其中一个搜索分支，并且将不满足列约束和对角线约束的方案都进行了剪枝。

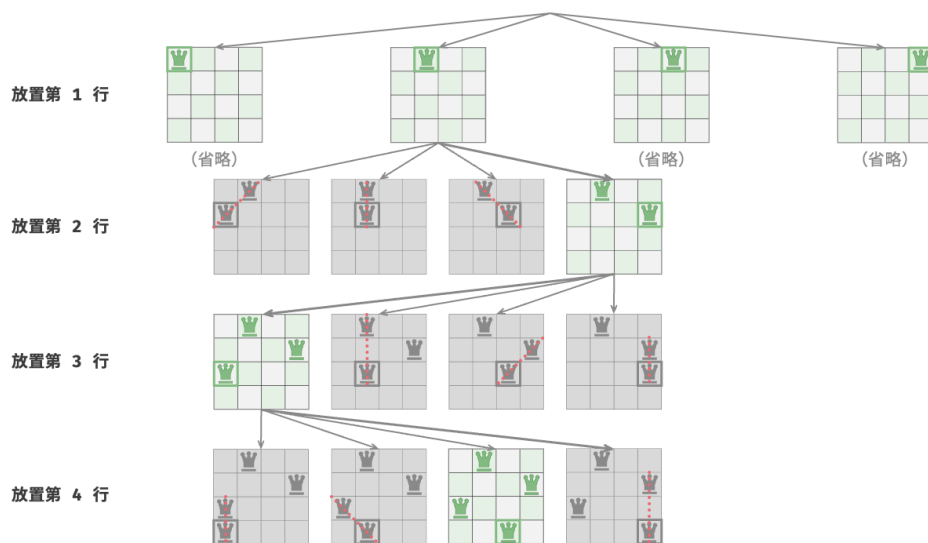


图 13-17 逐行放置策略

从本质上看，**逐行放置策略起到了剪枝的作用**，它避免了同一行出现多个皇后的所有搜索分支。

## 2. 列与对角线剪枝

为了满足列约束，我们可以利用一个长度为  $n$  的布尔型数组 `cols` 记录每一列是否有皇后。在每次决定放置前，我们通过 `cols` 将已有皇后的列进行剪枝，并在回溯中动态更新 `cols` 的状态。

### Tip

请注意，矩阵的起点位于左上角，其中行索引从上到下增加，列索引从左到右增加。

那么，如何处理对角线约束呢？设棋盘某个格子的行列索引为  $(row, col)$ ，选定矩阵中的某条主对角线，我们发现该对角线上所有格子的行索引减列索引都相等，即主对角线上所有格子的  $row - col$  为恒定值。

也就是说，如果两个格子满足  $row_1 - col_1 = row_2 - col_2$ ，则它们一定处在同一条主对角线上。利用该规律，我们可以借助图 13-18 所示的数组 `diags1` 记录每条主对角线上是否有皇后。

同理，次对角线上的所有格子的  $row + col$  是恒定值。我们同样也可以借助数组 `diags2` 来处理次对角线约束。

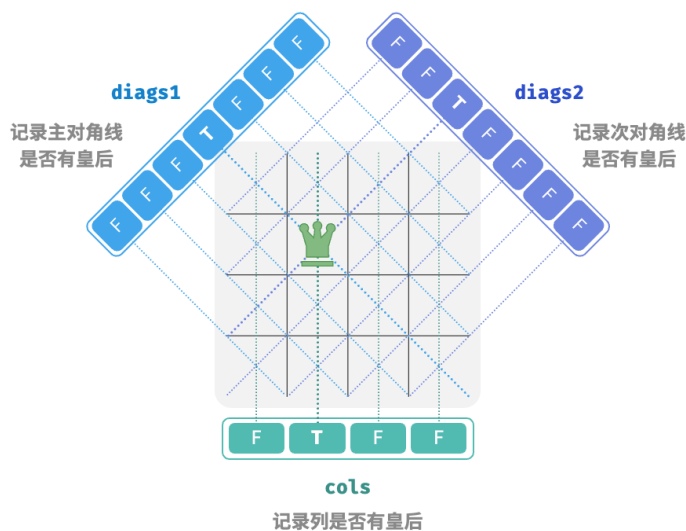


图 13-18 处理列约束和对角线约束

## 3. 代码实现

请注意， $n$  维方阵中  $row - col$  的范围是  $[-n + 1, n - 1]$ ， $row + col$  的范围是  $[0, 2n - 2]$ ，所以主对角线和次对角线的数量都为  $2n - 1$ ，即数组 `diags1` 和 `diags2` 的长度都为  $2n - 1$ 。

```
// === File: n_queens.c ===

/* 回溯算法：n 皇后 */
void backtrack(int row, int n, char state[MAX_SIZE][MAX_SIZE], char ***res, int *resSize, bool
↪ cols[MAX_SIZE],
    bool diags1[2 * MAX_SIZE - 1], bool diags2[2 * MAX_SIZE - 1]) {
```

```

// 当放置完所有行时，记录解
if (row == n) {
    res[*resSize] = (char **)malloc(sizeof(char *) * n);
    for (int i = 0; i < n; ++i) {
        res[*resSize][i] = (char *)malloc(sizeof(char) * (n + 1));
        strcpy(res[*resSize][i], state[i]);
    }
    (*resSize)++;
    return;
}

// 遍历所有列
for (int col = 0; col < n; col++) {
    // 计算该格子对应的主对角线和次对角线
    int diag1 = row - col + n - 1;
    int diag2 = row + col;
    // 剪枝：不允许该格子所在列、主对角线、次对角线上存在皇后
    if (!cols[col] && !diags1[diag1] && !diags2[diag2]) {
        // 尝试：将皇后放置在该格子
        state[row][col] = 'Q';
        cols[col] = diags1[diag1] = diags2[diag2] = true;
        // 放置下一行
        backtrack(row + 1, n, state, res, resSize, cols, diags1, diags2);
        // 回退：将该格子恢复为空位
        state[row][col] = '#';
        cols[col] = diags1[diag1] = diags2[diag2] = false;
    }
}
}

/* 求解 n 皇后 */
char ***nQueens(int n, int *returnSize) {
    char state[MAX_SIZE][MAX_SIZE];
    // 初始化 n*n 大小的棋盘，其中 'Q' 代表皇后，'#' 代表空位
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            state[i][j] = '#';
        }
        state[i][n] = '\0';
    }
    bool cols[MAX_SIZE] = {false}; // 记录列是否有皇后
    bool diags1[2 * MAX_SIZE - 1] = {false}; // 记录主对角线上是否有皇后
    bool diags2[2 * MAX_SIZE - 1] = {false}; // 记录次对角线上是否有皇后

    char ***res = (char ***)malloc(sizeof(char **) * MAX_SIZE);
    *returnSize = 0;
    backtrack(0, n, state, res, returnSize, cols, diags1, diags2);
    return res;
}

```

```
}
```

逐行放置  $n$  次，考虑列约束，则从第一行到最后一行分别有  $n$ 、 $n - 1$ 、...、2、1 个选择，使用  $O(n!)$  时间。当记录解时，需要复制矩阵 `state` 并添加进 `res`，复制操作使用  $O(n^2)$  时间。因此，**总体时间复杂度为  $O(n! \cdot n^2)$** 。实际上，根据对角线约束的剪枝也能够大幅缩小搜索空间，因而搜索效率往往优于以上时间复杂度。

数组 `state` 使用  $O(n^2)$  空间，数组 `cols`、`diags1` 和 `diags2` 皆使用  $O(n)$  空间。最大递归深度为  $n$ ，使用  $O(n)$  栈帧空间。因此，**空间复杂度为  $O(n^2)$** 。

## 13.5 小结

### 1. 重点回顾

- 回溯算法本质是穷举法，通过对解空间进行深度优先遍历来寻找符合条件的解。在搜索过程中，遇到满足条件的解则记录，直至找到所有解或遍历完成后结束。
- 回溯算法的搜索过程包括尝试与回退两个部分。它通过深度优先搜索来尝试各种选择，当遇到不满足约束条件的情况时，则撤销上一步的选择，退回到之前的状态，并继续尝试其他选择。尝试与回退是两个方向相反的操作。
- 回溯问题通常包含多个约束条件，它们可用于实现剪枝操作。剪枝可以提前结束不必要的搜索分支，大幅提升搜索效率。
- 回溯算法主要可用于解决搜索问题和约束满足问题。组合优化问题虽然可以用回溯算法解决，但往往存在效率更高或效果更好的解法。
- 全排列问题旨在搜索给定集合元素的所有可能的排列。我们借助一个数组来记录每个元素是否被选择，剪掉重复选择同一元素的搜索分支，确保每个元素只被选择一次。
- 在全排列问题中，如果集合中存在重复元素，则最终结果会出现重复排列。我们需要约束相等元素在每轮中只能被选择一次，这通常借助一个哈希集合来实现。
- 子集和问题的目标是在给定集合中找到和为目标值的所有子集。集合不区分元素顺序，而搜索过程会输出所有顺序的结果，产生重复子集。我们在回溯前将数据进行排序，并设置一个变量来指示每一轮的遍历起始点，从而将生成重复子集的搜索分支进行剪枝。
- 对于子集和问题，数组中的相等元素会产生重复集合。我们利用数组已排序的前置条件，通过判断相邻元素是否相等实现剪枝，从而确保相等元素在每轮中只能被选中一次。
- $n$  皇后问题旨在寻找将  $n$  个皇后放置到  $n \times n$  尺寸棋盘上的方案，要求所有皇后两两之间无法攻击对方。该问题的约束条件有行约束、列约束、主对角线和次对角线约束。为满足行约束，我们采用按行放置的策略，保证每一行放置一个皇后。
- 列约束和对角线约束的处理方式类似。对于列约束，我们利用一个数组来记录每一列是否有皇后，从而指示选中的格子是否合法。对于对角线约束，我们借助两个数组来分别记录该主、次对角线上是否存在皇后；难点在于找处在同一主（副）对角线上格子满足的行列索引规律。

### 2. Q&A

Q：怎么理解回溯和递归的关系？

总的来看，回溯是一种“算法策略”，而递归更像是一个“工具”。

- 回溯算法通常基于递归实现。然而，回溯是递归的应用场景之一，是递归在搜索问题中的应用。
- 递归的结构体现了“子问题分解”的解题范式，常用于解决分治、回溯、动态规划（记忆化递归）等问题。