递归、搜索与回溯算法

递归

> 在解决一个规模为n的问题时，如果满足以下条件，我们可以使用递归来解决：
>
> a. 问题可以被划分为规模更小的子问题，并且这些子问题具有与原问题相同的解决方法。
>
> b. 当我们知道规模更小的子问题（规模为 n - 1）的解时，我们可以直接计算出规模为 n 的问题的解。
>
> c. 存在一种简单情况，或者说当问题的规模足够小时，我们可以直接求解问题。
>
> 一般的递归求解过程如下：
>
> a. 验证是否满足简单情况。
>
> b. 假设较小规模的问题已经解决，解决当前问题。
>
> 上述步骤可以通过数学归纳法来证明。

1. 汉诺塔 （easy）

1. **题目链接：面试题 08.06. 汉诺塔问题**
2. **题目描述：**

> 在经典汉诺塔问题中，有 3 根柱子及 N 个不同大小的穿孔圆盘，盘子可以滑入任意一根柱子。一开始，所有盘子自上而下按升序依次套在第一根柱子上(即每一个盘子只能放在更大的盘子上面)。移动圆盘时受到以下限制:
>
> (1) 每次只能移动一个盘子;
>
> (2) 盘子只能从柱子顶端滑出移到下一根柱子;
>
> (3) 盘子只能叠在比它大的盘子上。
>
> 请编写程序，用栈将所有盘子从第一根柱子移到最后一根柱子。
>
> 你需要原地修改栈。
>
> 示例1:
>
> 输入：A = [2, 1, 0], B = [], C = []
>
> 输出：C = [2, 1, 0]

> 示例2:
>
> 输入：A = [1, 0], B = [], C = []
>
> 输出：C = [1, 0]
>
> 提示:
>
> A中盘子的数目不大于14个。

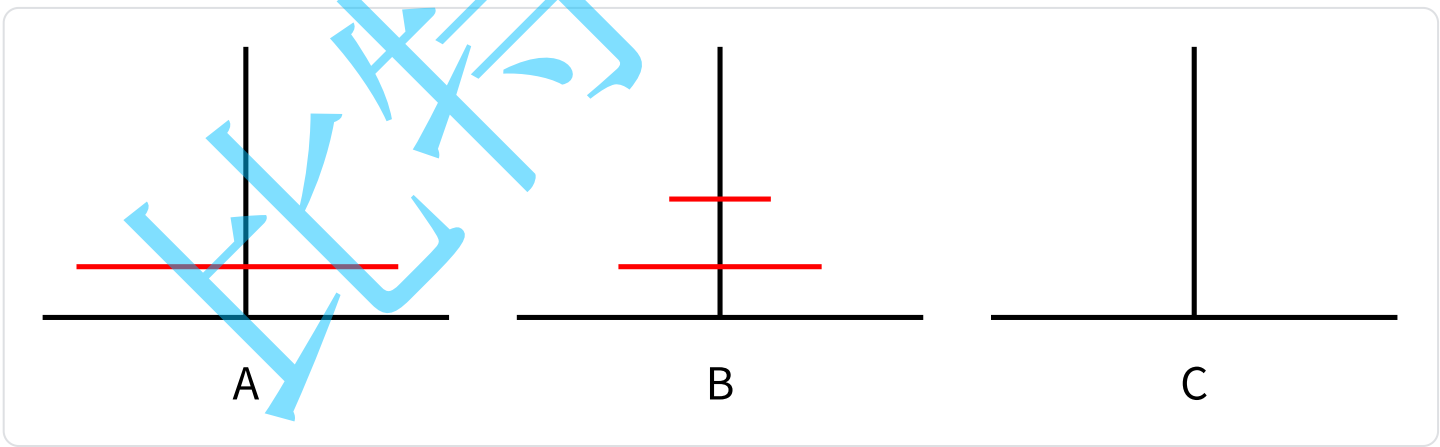
3. **解法（递归）：**

算法思路：

这是一道递归方法的经典题目，我们可以先从最简单的情况考虑：

- 假设 n = 1，只有一个盘子，很简单，直接把它从 A 中拿出来，移到 C 上；
- 如果 n = 2 呢？这时候我们就要借助 B 了，因为小盘子必须时刻都在大盘子上面，共需要 3 步（为了方便叙述，记 A 中的盘子从上到下为 1 号，2 号）：
  a. 1 号盘子放到 B 上；
  b. 2 号盘子放到 C 上；
  c. 1 号盘子放到 C 上。

  至此，C 中的盘子从上到下为 1 号，2 号。
- 如果 n > 2 呢？这是我们需要用到 n = 2 时的策略，将 A 上面的两个盘子挪到 B 上，再将最大的盘子挪到 C 上，最后将 B 上的小盘子挪到 C 上就完成了所有步骤。例如 n = 3 时如下图：



因为 A 中最后处理的是最大的盘子，所以在移动过程中不存在大盘子在小盘子上面的情况。

则本题可以被解释为：

1. 对于规模为 n 的问题，我们需要将 A 柱上的 n 个盘子移动到C柱上。
2. 规模为 n 的问题可以被拆分为规模为 n-1 的子问题：
   a. 将 A 柱上的上面 n-1 个盘子移动到B柱上。
   b. 将 A 柱上的最大盘子移动到 C 柱上，然后将 B 柱上的 n-1 个盘子移动到C柱上。

3. 当问题的规模变为 n=1 时，即只有一个盘子时，我们可以直接将其从 A 柱移动到 C 柱。

- 需要注意的是，步骤 2.b 考虑的是总体问题中的 子问题b 情况。在处理子问题的 子问题b 时，我们应该将 A 柱中的最上面的盘子移动到 C 柱，然后再将 B 柱上的盘子移动到 C 柱。在处理总体问题的 子问题b 时，A 柱中的最大盘子仍然是最上面的盘子，因此这种做法是通用的。

算法流程：

递归函数设计：void hanotaa(vector<int>& A, vector<int>& B, vector<int>& C, int n)

1. 返回值：无；
2. 参数：三个柱子上的盘子，当前需要处理的盘子个数（当前问题规模）。
3. 函数作用：将 A 中的上面 n 个盘子挪到 C 中。

递归函数流程：

1. 当前问题规模为 n=1 时，直接将 A 中的最上面盘子挪到 C 中并返回；
2. 递归将 A 中最上面的 n-1 个盘子挪到 B 中；
3. 将 A 中最上面的一个盘子挪到 C 中；
4. 将 B 中上面 n-1 个盘子挪到 C 中。

C++ 算法代码：

```cpp
class Solution
{
public:
    void hanota(vector<int>& a, vector<int>& b, vector<int>& c)
    {
        dfs(a, b, c, a.size());
    }

    void dfs(vector<int>& a, vector<int>& b, vector<int>& c, int n)
    {
        if(n == 1)
        {
            c.push_back(a.back());
            a.pop_back();
            return;
        }

        dfs(a, c, b, n - 1);
        c.push_back(a.back());
```

```
        a.pop_back();
        dfs(b, a, c, n - 1);
    }
};
```

C++ 运行结果：

Java 算法代码：

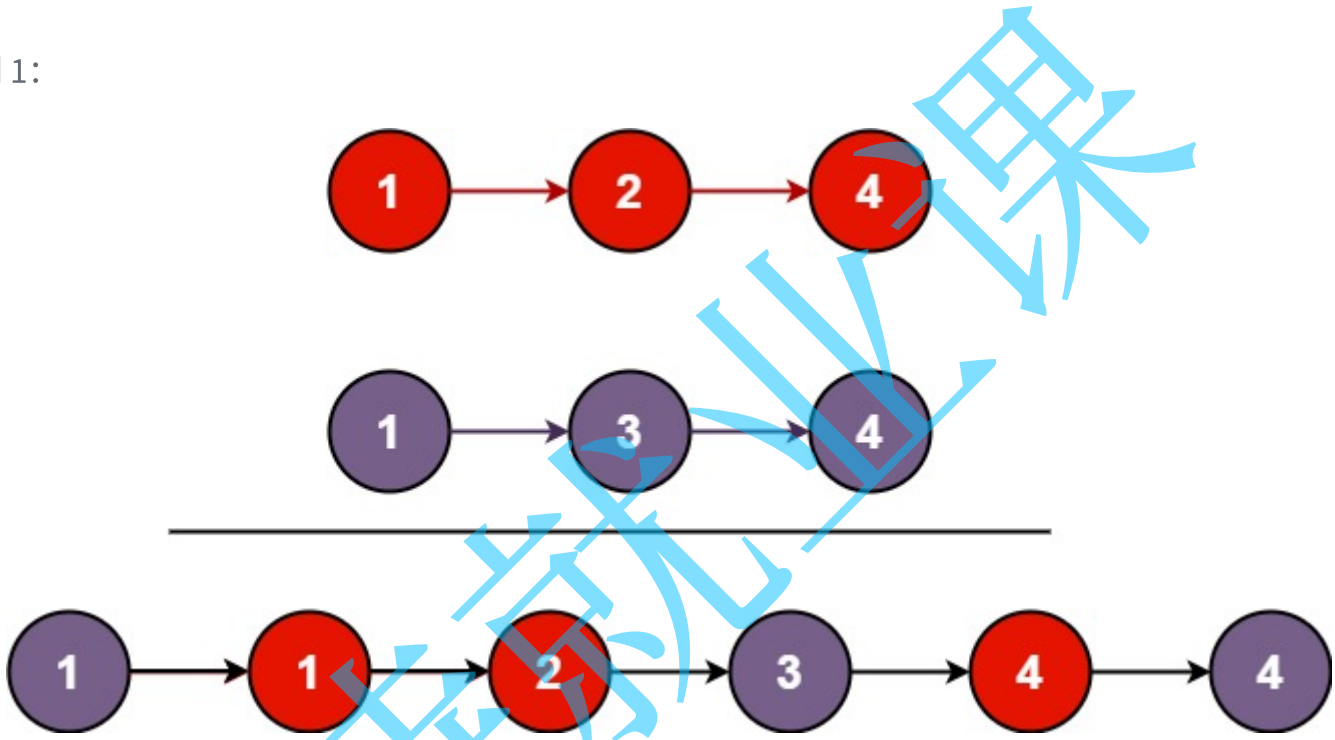
```java
class Solution
{
    public void hanota(List<Integer> a, List<Integer> b, List<Integer> c)
    {
        dfs(a, b, c, a.size());
    }

    public void dfs(List<Integer> a, List<Integer> b, List<Integer> c, int n)
    {
        if(n == 1)
        {
            c.add(a.remove(a.size() - 1));
            return;
        }

        dfs(a, c, b, n - 1);
        c.add(a.remove(a.size() - 1));
        dfs(b, a, c, n - 1);
    }
}
```

Java 运行结果：

2. 合并两个有序链表（easy）

1. **题目链接：21. 合并两个有序链表**

2. **题目描述：**

> 将两个升序链表合并为一个新的 升序 链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。
>
> 示例 1：
>
> 
>
>
> 输入：l1 = [1,2,4], l2 = [1,3,4]
>
> 输出：[1,1,2,3,4,4]
>
> 示例 2：
>
> 输入：l1 = [], l2 = []
>
> 输出：[]
>
> 示例 3：
>
> 输入：l1 = [], l2 = [0]
>
> 输出：[0]
>
> 提示：
>
> 两个链表的节点数目范围是 [0, 50]
>
> -100 <= Node.val <= 100
>
> l1 和 l2 均按 非递减顺序 排列

3. **解法（递归）：**

算法思路：

1. **递归函数的含义**：交给你两个链表的头结点，你帮我把它们合并起来，并且返回合并后的头结点；
2. **函数体**：选择两个头结点中较小的结点作为最终合并后的头结点，然后将剩下的链表交给递归函数去处理；
3. **递归出口**：当某一个链表为空的时候，返回另外一个链表。

注意注意注意：链表的题一定要画图，搞清楚指针的操作！

C++ 算法代码：

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution
{
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2)
    {
        if(l1 == nullptr) return l2;
        if(l2 == nullptr) return l1;

        if(l1->val <= l2->val)
        {
            l1->next = mergeTwoLists(l1->next, l2);
            return l1;
        }
        else
        {
            l2->next = mergeTwoLists(l1, l2->next);
            return l2;
        }
    }
};
```

C++ 代码结果：

Java 算法代码：

```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution
{
    public ListNode mergeTwoLists(ListNode l1, ListNode l2)
    {
        if(l1 == null) return l2;
        if(l2 == null) return l1;

        if(l1.val <= l2.val)
        {
            l1.next = mergeTwoLists(l1.next, l2);
            return l1;
        }
        else
        {
            l2.next = mergeTwoLists(l1, l2.next);
            return l2;
        }
    }
}
```


Java 运行结果：

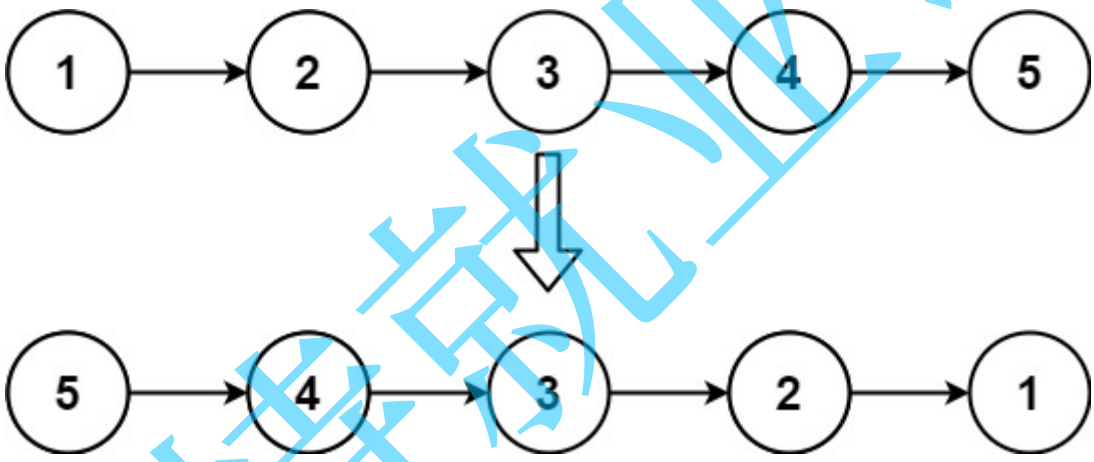
3. 反转链表（easy）

1. **题目链接：206. 反转链表**

2. **题目描述：**

给你单链表的头节点 head ，请你反转链表，并返回反转后的链表。

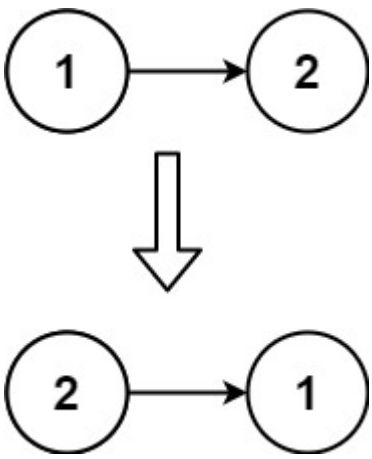
示例 1：



输入：head = [1,2,3,4,5]

输出：[5,4,3,2,1]

示例 2：



输入：head = [1,2]

输出：[2,1]

> 示例 3：
>
> 输入：head = []
>
> 输出：[]
>
> 提示：
>
> 链表中节点的数目范围是 [0, 5000]
>
> -5000 <= Node.val <= 5000
>
> 进阶：链表可以选用迭代或递归方式完成反转。你能否用两种方法解决这道题？

3. **解法（递归）：**

算法思路：

1. **递归函数的含义**：交给你一个链表的头指针，你帮我逆序之后，返回逆序后的头结点；
2. **函数体**：先把当前结点之后的链表逆序，逆序完之后，把当前结点添加到逆序后的链表后面即可；
3. **递归出口**：当前结点为空或者当前只有一个结点的时候，不用逆序，直接返回。

注意注意注意：链表的题一定要画图，搞清楚指针的操作！

C++ 算法代码：

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution
{
public:
    ListNode* reverseList(ListNode* head)
    {
        if(head == nullptr || head->next == nullptr) return head;

        ListNode* newHead = reverseList(head->next);
        head->next->next = head;
```

```
        head->next = nullptr;

        return newHead;
    }
};
```

C++ 代码结果：

Java 算法代码：

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution
{
    public ListNode reverseList(ListNode head)
    {
        if(head == null || head.next == null) return head;

        ListNode newHead = reverseList(head.next);
        head.next.next = head;
        head.next = null;

        return newHead;
    }
}
```

Java 运行结果：

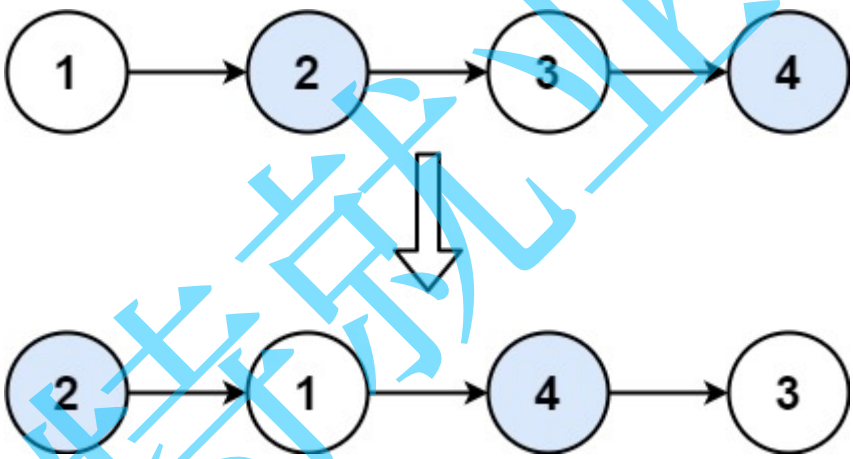
4. 两两交换链表中的节点（medium）

1. **题目链接：24. 两两交换链表中的节点**

2. **题目描述：**

给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头节点。你必须在不修改节点内部的值的情况下完成本题（即，只能进行节点交换）。

示例 1：



输入：head = [1,2,3,4]

输出：[2,1,4,3]

示例 2：

输入：head = []

输出：[]

示例 3：

输入：head = [1]

输出：[1]

提示：

链表中节点的数目在范围 [0, 100] 内

0 <= Node.val <= 100

3. **解法（递归）：**

算法思路：

1. **递归函数的含义**：交给你一个链表，将这个链表两两交换一下，然后返回交换后的头结点；
2. **函数体**：先去处理一下第二个结点往后的链表，然后再把当前的两个结点交换一下，连接上后面处理后的链表；
3. **递归出口**：当前结点为空或者当前只有一个结点的时候，不用交换，直接返回。

注意注意注意：链表的题一定要画图，搞清楚指针的操作！

C++ 算法代码：

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution
{
public:
    ListNode* swapPairs(ListNode* head)
    {
        if(head == nullptr || head->next == nullptr) return head;

        auto tmp = swapPairs(head->next->next);
        auto ret = head->next;
        head->next->next = head;
        head->next = tmp;

        return ret;
    }
};
```

C++ 代码结果：

Java 算法代码：

```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution
{
    public ListNode swapPairs(ListNode head)
    {
        if(head == null || head.next == null) return head;

        ListNode tmp = swapPairs(head.next.next);
        ListNode ret = head.next;
        ret.next = head;
        head.next = tmp;

        return ret;
    }
}
```

Java 运行结果：

5. Pow（x, n）- 快速幂（medium）

1. **题目链接：50. Pow(x, n)**

2. **题目描述：**

> 实现 pow(x, n) ，即计算 x 的整数 n 次幂函数（即，x^n ）。
>
> 示例 1:
>
> 输入：x = 2.00000, n = 10
>
> 输出：1024.00000
>
> 示例 2:
>
> 输入：x = 2.10000, n = 3
>
> 输出：9.26100
>
> 示例 3:
>
> 输入：x = 2.00000, n = -2
>
> 输出：0.25000
>
> 解释：2-2 = 1/22 = 1/4 = 0.25
>
> 提示:
>
> -100.0 < x < 100.0
>
> -2^31 <= n <= 2^31-1
>
> n 是一个整数
>
> 要么 x 不为零，要么 n > 0 。
>
> -10^4 <= x^n <= 10^4

3. **解法（递归 - 快速幂）：**

算法思路：

1. **递归函数的含义**：求出 `x` 的 `n` 次方是多少，然后返回；
2. **函数体**：先求出 `x` 的 `n / 2` 次方是多少，然后根据 `n` 的奇偶，得出 `x` 的 `n` 次方是多少；
3. **递归出口**：当 `n` 为 `0` 的时候，返回 `1` 即可。

C++ 算法代码：

```
1 class Solution
2 {
```

```
public:
    double myPow(double x, int n)
    {
        return n < 0 ? 1.0 / pow(x, -(long long)n) : pow(x, n);
    }

    double pow(double x, long long n)
    {
        if(n == 0) return 1.0;
        double tmp = pow(x, n / 2);
        return n % 2 == 0 ? tmp * tmp : tmp * tmp * x;
    }
};
```

C++ 代码结果：

Java 算法代码：

```
class Solution
{
    public double myPow(double x, int n)
    {
        return n < 0 ? 1.0 / pow(x, -n) : pow(x, n);
    }

    public double pow(double x, int n)
    {
        if(n == 0) return 1.0;
        double tmp = pow(x, n / 2);
        return n % 2 == 0 ? tmp * tmp : tmp * tmp * x;
    }
}
```

Java 运行结果：

二叉树中的深搜

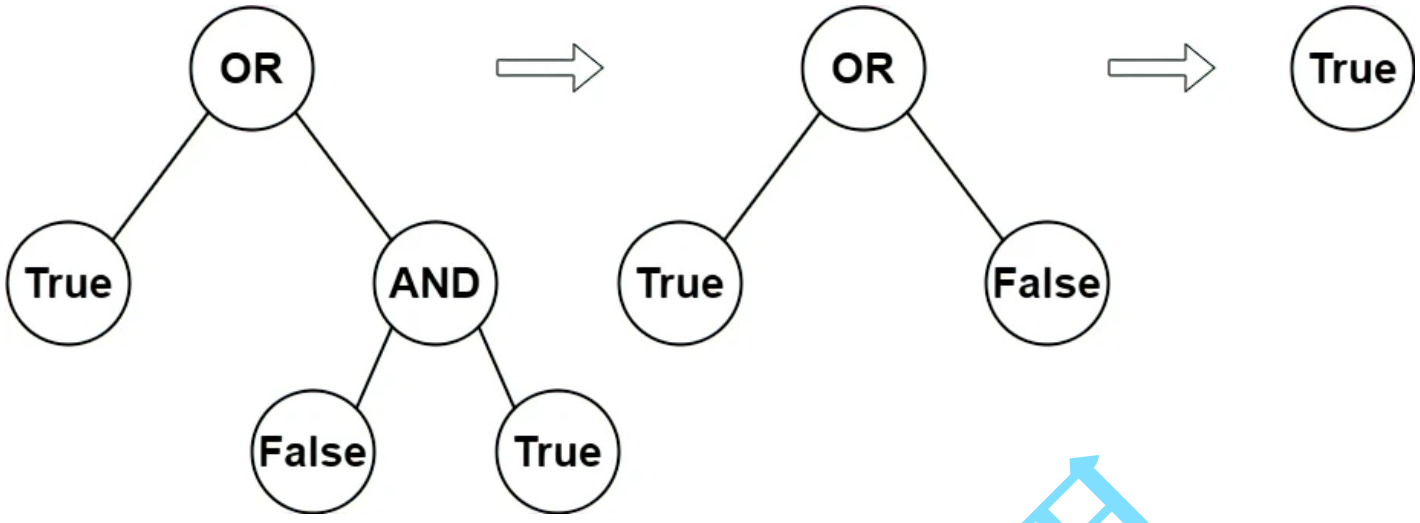
> 深度优先遍历（**DFS**，全称为 **Depth First Traversal**），是我们树或者图这样的数据结构中常用的一种**遍历算法**。这个算法会尽可能深的搜索树或者图的分支，直到一条路径上的所有节点都被遍历完毕，然后再回溯到上一层，继续找一条路遍历。
>
> 在二叉树中，常见的深度优先遍历为：**前序遍历**、**中序遍历**以及**后序遍历**。
>
> 因为树的定义本身就是递归定义，因此采用递归的方法去实现树的三种遍历不仅容易理解而且代码很简洁。并且前中后序三种遍历的唯一区别就是**访问根节点的时机不同**，在做题的时候，选择一个适当的遍历顺序，对于算法的理解是非常有帮助的。

6. 计算布尔二叉树的值（medium）

1. **题目链接：2331. 计算布尔二叉树的值**
2. **题目描述：**

> 给你一棵 完整二叉树 的根，这棵树有以下特征：
>
> 叶子节点 要么值为 0 要么值为 1 ，其中 0 表示 False ，1 表示 True 。
>
> 非叶子节点 要么值为 2 要么值为 3 ，其中 2 表示逻辑或 OR ，3 表示逻辑与 AND 。
>
> 计算 一个节点的值方式如下：
>
> 如果节点是个叶子节点，那么节点的 值 为它本身，即 True 或者 False 。
>
> 否则，计算 两个孩子的节点值，然后将该节点的运算符对两个孩子值进行 运算 。
>
> 返回根节点 root 的布尔运算值。
>
> 完整二叉树 是每个节点有 0 个或者 2 个孩子的二叉树。
>
> 叶子节点 是没有孩子的节点。

- 示例 1：



输入：root = [2,1,3,null,null,0,1]

输出：true

解释：上图展示了计算过程。

AND 与运算节点的值为 False AND True = False 。

OR 运算节点的值为 True OR False = True 。

根节点的值为 True ，所以我们返回 true 。

- 示例 2：

输入：root = [0]

输出：false

解释：根节点是叶子节点，且值为 false，所以我们返回 false 。

- 提示：

树中节点数目在 [1, 1000] 之间。

0 <= Node.val <= 3

每个节点的孩子数为 0 或 2 。

叶子节点的值为 0 或 1 。

非叶子节点的值为 2 或 3 。

3. **解法（递归）：**

算法思路：

算法思路：

本题可以被解释为：

1. 对于规模为 n 的问题，需要求得当前节点值。
2. 节点值不为 0 或 1 时，规模为 n 的问题可以被拆分为规模为 n-1 的子问题：
    a. 所有子节点的值；

b. 通过子节点的值运算出当前节点值。

3. 当问题的规模变为 n=1 时，即叶子节点的值为 0 或 1，我们可以直接获取当前节点值为 0 或 1。

算法流程：

递归函数设计：bool evaluateTree(TreeNode* root)

1. 返回值：当前节点值；
2. 参数：当前节点指针。
3. 函数作用：求得当前节点通过逻辑运算符得出的值。

递归函数流程：

1. 当前问题规模为 n=1 时，即叶子节点，直接返回当前节点值；
2. 递归求得左右子节点的值；
3. 通过判断当前节点的逻辑运算符，计算左右子节点值运算得出的结果；

C++ 算法代码：

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution
{
public:
    bool evaluateTree(TreeNode* root)
    {
        if(root->left == nullptr) return root->val == 0 ? false : true;

        bool left = evaluateTree(root->left);
        bool right = evaluateTree(root->right);
        return root->val == 2 ? left | right : left & right;
```

```
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution
{
    public boolean evaluateTree(TreeNode root)
    {
        if(root.left == null) return root.val == 0 ? false : true;

        boolean left = evaluateTree(root.left);
        boolean right = evaluateTree(root.right);
        return root.val == 2 ? left | right : left & right;
    }
}
```

Java 运行结果：

7. 求根节点到叶节点数字之和（medium）

1. **题目链接：129. 求根节点到叶节点数字之和**

2. **题目描述：**

给你一个二叉树的根节点 root ，树中每个节点都存放有一个 0 到 9 之间的数字。

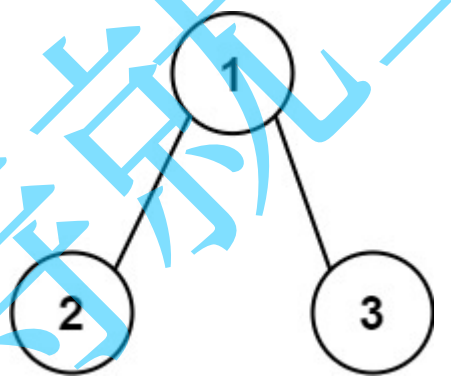
每条从根节点到叶节点的路径都代表一个数字：

例如，从根节点到叶节点的路径 1 -> 2 -> 3 表示数字 123 。

计算从根节点到叶节点生成的 所有数字之和 。

叶节点 是指没有子节点的节点。

示例 1：



输入：root = [1,2,3]

输出：25

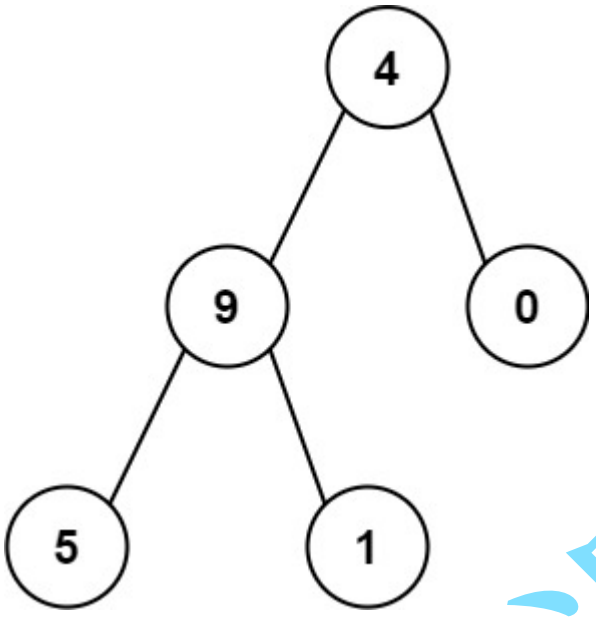
解释：

从根到叶子节点路径 1->2 代表数字 12

从根到叶子节点路径 1->3 代表数字 13

因此，数字总和 = 12 + 13 = 25

示例 2：



输入：root = [4,9,0,5,1]

输出：1026

解释：

从根到叶子节点路径 4->9->5 代表数字 495

从根到叶子节点路径 4->9->1 代表数字 491

从根到叶子节点路径 4->0 代表数字 40

因此，数字总和 = 495 + 491 + 40 = 1026

3. **解法（dfs - 前序遍历）：**

前序遍历按照根节点、左子树、右子树的顺序遍历二叉树的所有节点，通常用于子节点的状态依赖于父节点状态的题目。

算法思路：

在**前序遍历**的过程中，我们可以往左右子树传递信息，并且在回溯时得到左右子树的返回值。递归函数可以帮我们完成两件事：

1. 将父节点的数字与当前节点的信息整合到一起，计算出当前节点的数字，然后传递到下一层进行递归；
2. 当遇到叶子节点的时候，就不再向下传递信息，而是将整合的结果向上一直回溯到根节点。

在递归结束时，根节点需要返回的值也就被更新为了整棵树的数字和。

算法流程：

递归函数设计：int dfs(TreeNode* root, int num)

1. 返回值：当前子树计算的结果（数字和）；

2. 参数 **num**：递归过程中往下传递的信息（父节点的数字）；
3. 函数作用：整合父节点的信息与当前**节点**的信息计算当前节点数字，并向下传递，在回溯时返回当前**子树（当前节点作为子树根节点）**数字和。

递归函数流程：

1. 当遇到空节点的时候，说明这条路从根节点开始没有分支，返回 **0**；
2. 结合父节点传下的信息以及当前节点的 **val**，计算出当前节点数字 **sum**；
3. 如果当前结点是叶子节点，直接返回整合后的结果 **sum**；
4. 如果当前结点不是叶子节点，将 **sum** 传到左右子树中去，得到左右子树中节点路径的数字和，然后相加后返回结果。

C++ 算法代码：

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution
{
public:
    int sumNumbers(TreeNode* root)
    {
        return dfs(root, 0);
    }

    int dfs(TreeNode* root, int presum)
    {
        presum = presum * 10 + root->val;
        if(root->left == nullptr && root->right == nullptr)
            return presum;
        int ret = 0;
        if(root->left) ret += dfs(root->left, presum);
        if(root->right) ret += dfs(root->right, presum);
```

```
        return ret;
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution
{
    public int sumNumbers(TreeNode root)
    {
        return dfs(root, 0);
    }

    public int dfs(TreeNode root, int preSum)
    {
        preSum = preSum * 10 + root.val;
        if(root.left == null && root.right == null)
            return preSum;

        int ret = 0;
        if(root.left != null) ret += dfs(root.left, preSum);
```

```
31          if(root.right != null) ret += dfs(root.right, preSum);
32          return ret;
33      }
34  }
```

Java 运行结果：

8. 二叉树剪枝（medium）

1. **题目链接：814. 二叉树剪枝**

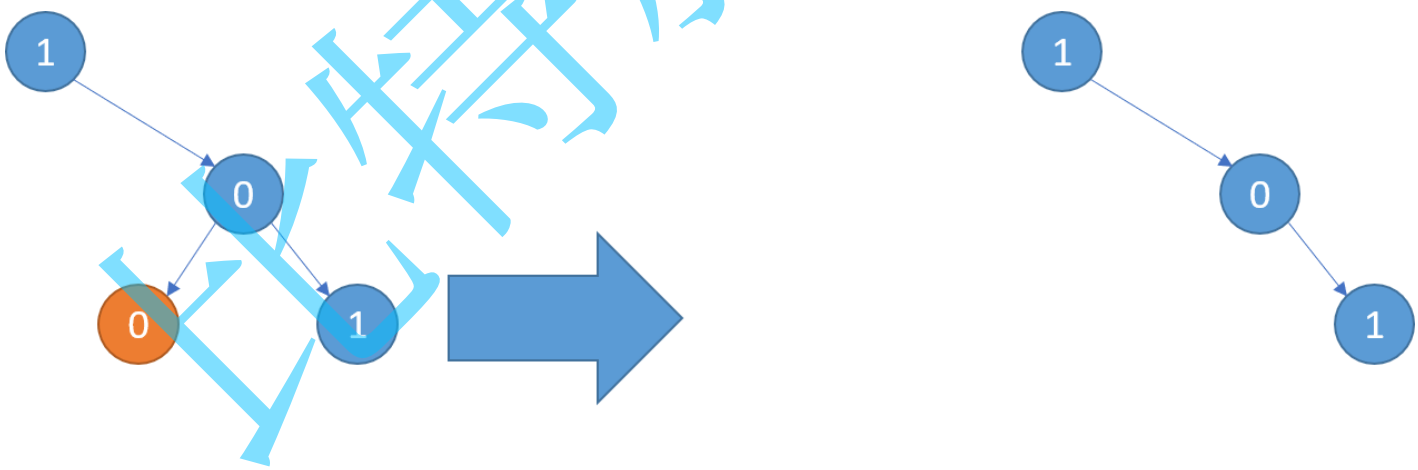
2. **题目描述：**

给你二叉树的根结点 root ，此外树的每个结点的值要么是 0 ，要么是 1 。

返回移除了所有不包含 1 的子树的原二叉树。

节点 node 的子树为 node 本身加上所有 node 的后代。

示例 1：

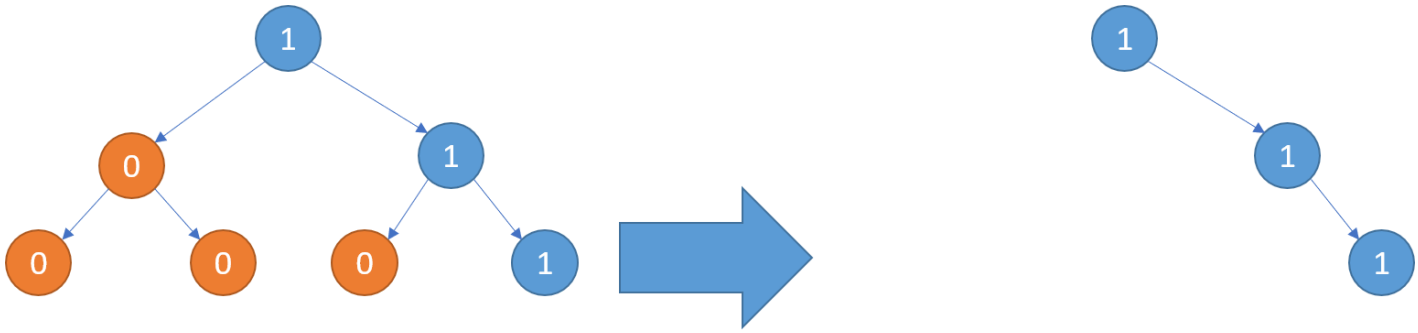


输入：root = [1,null,0,0,1]

输出：[1,null,0,null,1]

解释：只有红色节点满足条件“所有不包含 1 的子树”。 右图为返回的答案。

示例 2：



输入：root = [1,0,1,0,0,0,1]

输出：[1,null,1,null,1]

3. **解法（dfs - 后序遍历）：**

后序遍历按照左子树、右子树、根节点的顺序遍历二叉树的所有节点，通常用于父节点的状态依赖于子节点状态的题目。

算法思路：

如果我们选择从上往下删除，我们需要收集左右子树的信息，这可能导致代码编写相对困难。然而，通过观察我们可以发现，如果我们先删除最底部的叶子节点，然后再处理删除后的节点，最终的结果并不会受到影响。

因此，我们可以采用后序遍历的方式来解决这个问题。在后序遍历中，我们先处理左子树，然后处理右子树，最后再处理当前节点。在处理当前节点时，我们可以判断其是否为叶子节点且其值是否为 0，如果满足条件，我们可以删除当前节点。

- 需要注意的是，在删除叶子节点时，其父节点很可能会成为新的叶子节点。因此，在处理完子节点后，我们仍然需要处理当前节点。这也是为什么选择后序遍历的原因（后序遍历首先遍历到的一定是叶子节点）。
- 通过使用后序遍历，我们可以逐步删除叶子节点，并且保证删除后的节点仍然满足删除操作的要求。这样，我们可以较为方便地实现删除操作，而不会影响最终的结果。
- 若在处理结束后所有叶子节点的值均为 1，则所有子树均包含 1，此时可以返回。

算法流程：

递归函数设计：void dfs(TreeNode*& root)

1. 返回值：无；
2. 参数 ： 当前需要处理的节点；
3. 函数作用：判断当前节点是否需要删除，若需要删除，则删除当前节点。

后序遍历的主要流程：

1. 递归出口：当传入节点为空时，不做任何处理；
2. 递归处理左子树；
3. 递归处理右子树；
4. 处理当前节点：判断该节点是否为叶子节点（即左右子节点均被删除，当前节点成为叶子节点），并且节点的值为 0：
   a. 如果是，就删除掉；
   b. 如果不是，就不做任何处理。

C++ 算法代码：

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution
{
public:
    TreeNode* pruneTree(TreeNode* root)
    {
        if(root == nullptr) return nullptr;

        root->left = pruneTree(root->left);
        root->right = pruneTree(root->right);
        if(root->left == nullptr && root->right == nullptr && root->val == 0)
        {
            delete root; // 防止内泄漏
            root = nullptr;
        }
        return root;
    }
};
```

C++ 运行结果：

Java 算法代码：

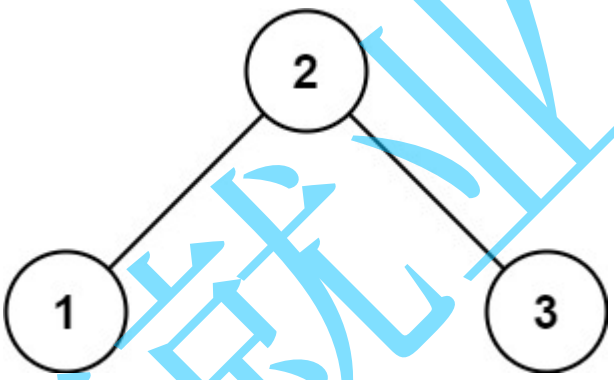
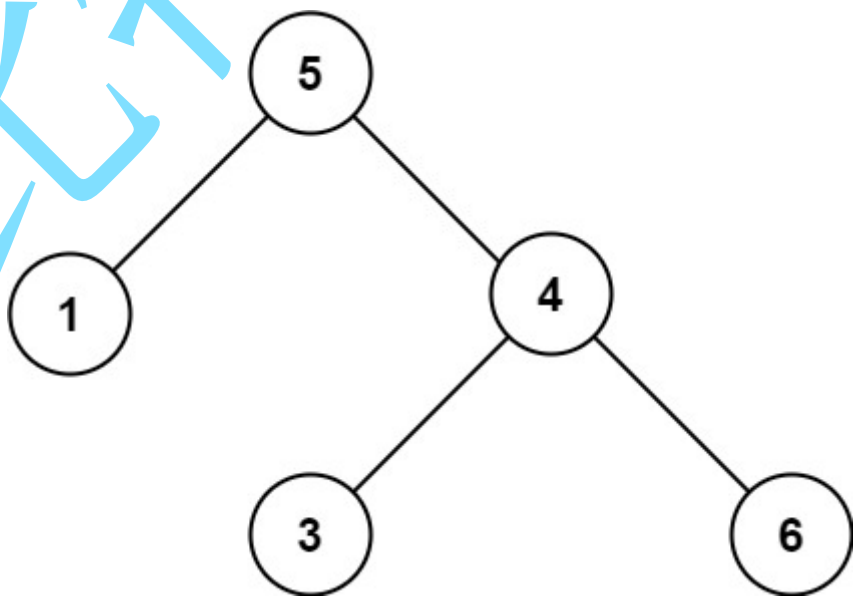
```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution
{
    public TreeNode pruneTree(TreeNode root)
    {
        if(root == null) return null;

        root.left = pruneTree(root.left);
        root.right = pruneTree(root.right);
        if(root.left == null && root.right == null && root.val == 0)
            root = null;
        return root;
    }
}
```


Java 运行结果：

9. 验证二叉搜索树（medium）

1. **题目链接：98. 验证二叉搜索树**

2. **题目描述：**

> 给你一个二叉树的根节点 root，判断其是否是一个有效的二叉搜索树。
>
> 有效 二叉搜索树定义如下：
>
> - 节点的左子树只包含 小于 当前节点的数。
> - 节点的右子树只包含 大于 当前节点的数。
> - 所有左子树和右子树自身必须也是二叉搜索树。
>
> 示例 1：
>
> 
>
>
> 输入：root = [2,1,3]
> 输出：true
>
> 示例 2：
>
> 
>
>
> 输入：root = [5,1,4,null,null,3,6]
> 输出：false
> 解释：根节点的值是 5 ，但是右子节点的值是 4 。

3. **解法（利用中序遍历）：**

后序遍历按照左子树、根节点、右子树的顺序遍历二叉树的所有节点，通常用于二叉搜索树相关题目。

算法思路：

如果一棵树是二叉搜索树，那么它的中序遍历的结果一定是一个严格递增的序列。

因此，我们可以初始化一个**无穷小**的全区变量，用来记录中序遍历过程中的**前驱结点**。那么就可以在中序遍历的过程中，先判断是否和前驱结点构成递增序列，然后修改前驱结点为当前结点，传入下一层的递归中。

算法流程：

1. 初始化一个全局的变量 **prev**，用来记录中序遍历过程中的前驱结点的 **val**；
2. **中序遍历的递归函数**中：
   a. 设置递归出口：**root == nullptr** 的时候，返回 **true**；
   b. 先递归判断左子树是否是二叉搜索树，用 **retleft** 标记；
   c. 然后判断当前结点是否满足二叉搜索树的性质，用 **retcur** 标记：
      - 如果当前结点的 **val** 大于 **prev**，说明满足条件，**retcur** 改为 **true**；
      - 如果当前结点的 **val** 小于等于 **prev**，说明不满足条件，**retcur** 改为 **false**；
   d. 最后递归判断右子树是否是二叉搜索树，用 **retright** 标记；
3. 只有当 **retleft**、 **retcur** 和 **retright** 都是 **true** 的时候，才返回 **true**。

C++ 算法代码：

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution
```

```
{
    long prev = LONG_MIN;
public:
    bool isValidBST(TreeNode* root)
    {
        if(root == nullptr) return true;
        bool left = isValidBST(root->left);
        // 剪枝
        if(left == false) return false;

        bool cur = false;
        if(root->val > prev)
            cur = true;
        // 剪枝
        if(cur == false) return false;

        prev = root->val;
        bool right = isValidBST(root->right);
        return left && right && cur;
    }
};
```

C++ 运行结果：

Java 算法代码：

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
```

```java
 *     }
 * }
 */
class Solution
{
    long prev = Long.MIN_VALUE;
    public boolean isValidBST(TreeNode root)
    {
        if(root == null) return true;

        boolean left = isValidBST(root.left);
        // 剪枝
        if(left == false) return false;

        boolean cur = false;
        if(root.val > prev) cur = true;
        if(cur == false) return false;

        prev = root.val;
        boolean right = isValidBST(root.right);
        return left && cur && right;
    }
}
```

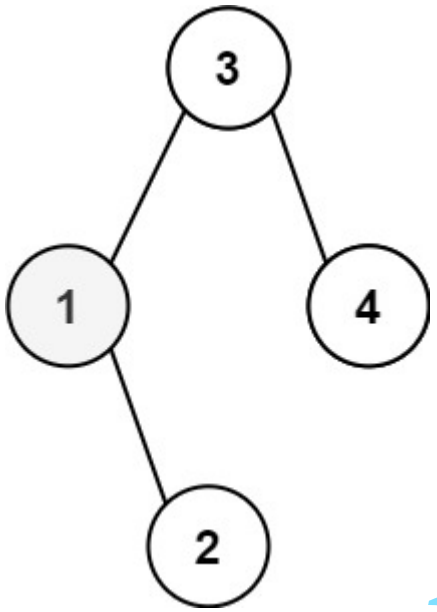
Java 运行结果：

10. 二叉搜索树中第 k 小的元素（medium）

1. **题目链接：230. 二叉搜索树中第 K 小的元素**

2. **题目描述：**

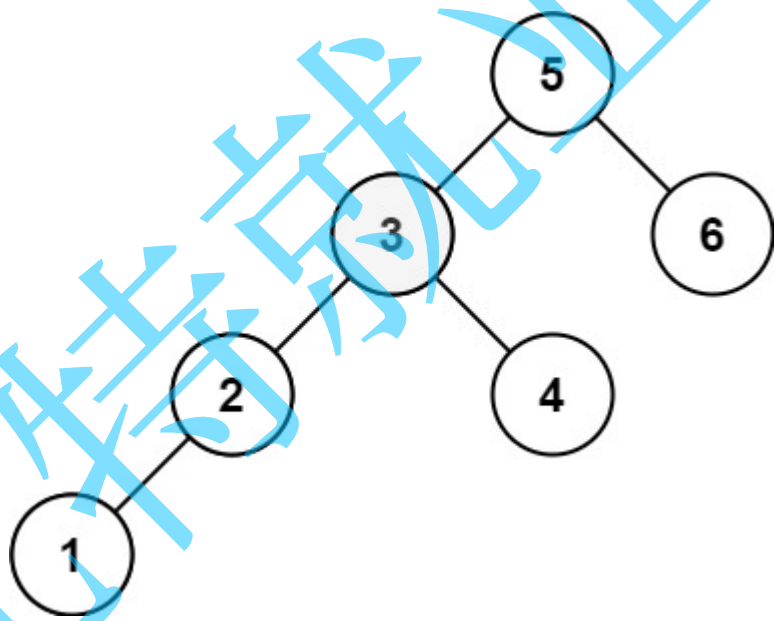
> 给定一个二叉搜索树的根节点 root ，和一个整数 k ，请你设计一个算法查找其中第 k 个最小元素（从 1 开始计数）。
>
> 示例 1：



输入：root = [3,1,4,null,2], k = 1

输出：1

示例 2：



输入：root = [5,3,6,2,4,null,null,1], k = 3

输出：3

3. **解法二（中序遍历 + 计数器剪枝）：**

算法思路：

上述解法不仅使用大量额外空间存储数据，并且会将所有的结点都遍历一遍。

但是，我们可以根据中序遍历的过程，只需扫描前 **k** 个结点即可。

因此，我们可以创建一个全局的计数器 **count**，将其初始化为 **k**，每遍历一个节点就将 **count--**。直到某次递归的时候，**count** 的值等于 **1**，说明此时的结点就是我们要找的结果。

算法流程：

1. 定义一个全局的变量 count，在主函数中初始化为 k 的值（不用全局也可以，当成参数传入递归过程中）；

递归函数的设计：int dfs(TreeNode* root)：

- 返回值为第 **k** 个结点；

递归函数流程（中序遍历）：

1. 递归出口：空节点直接返回 **-1**，说明没有找到；
2. 去左子树上查找结果，记为 **retleft**：
   a. 如果 **retleft == -1**，说明没找到，继续执行下面逻辑；
   b. 如果 **retleft != -1**，说明找到了，直接返回结果，无需执行下面代码（**剪枝**）；
3. 如果左子树没找到，判断当前结点是否符合：
   a. 如果符合，直接返回结果
4. 如果当前结点不符合，去右子树上寻找结果。

C++ 算法代码：

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution
{
    int count;
    int ret;
public:
    int kthSmallest(TreeNode* root, int k)
    {
        count = k;
```

```
        dfs(root);
        return ret;
    }

    void dfs(TreeNode* root)
    {
        if(root == nullptr || count == 0) return;
        dfs(root->left);
        count--;
        if(count == 0) ret = root->val;
        dfs(root->right);
    }

};
```

C++ 运行结果：

Java 算法代码：

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution
{
    int count;
    int ret;
```

```java
    public int kthSmallest(TreeNode root, int k)
    {
        count = k;
        dfs(root);
        return ret;
    }

    void dfs(TreeNode root)
    {
        if(root == null || count == 0) return;
        dfs(root.left);


        count--;
        if(count == 0) ret = root.val;

        if(count == 0) return;
        dfs(root.right);
    }
}
```

Java 运行结果：

11. 二叉树的所有路径（easy）

1. **题目链接：257. 二叉树的所有路径**
2. **题目描述：**

> 给你一个二叉树的根节点 root ，按 任意顺序 ，返回所有从根节点到叶子节点的路径。
>
> 叶子节点 是指没有子节点的节点。

- 示例 1：

> 输入：root = [1,2,3,null,5]
>
> 输出：["1->2->5","1->3"]

- 示例 2：

> 输入：root = [1]

> 输出：["1"]

- 提示：

> 树中节点的数目在范围 [1, 100] 内
>
> -100 <= Node.val <= 100

3. **解法（回溯）：**

算法思路：

使用深度优先遍历（DFS）求解。

路径以字符串形式存储，从根节点开始遍历，每次遍历时将当前节点的值加入到路径中，如果该节点为叶子节点，将路径存储到结果中。否则，将 "->" 加入到路径中并递归遍历该节点的左右子树。

定义一个结果数组，进行递归。递归具体实现方法如下：

1. 如果当前节点不为空，就将当前节点的值加入路径 path 中，否则直接返回；
2. 判断当前节点是否为叶子节点，如果是，则将当前路径加入到所有路径的存储数组 paths 中；
3. 否则，将当前节点值加上 "->" 作为路径的分隔符，继续递归遍历当前节点的左右子节点。
4. 返回结果数组。

- 特别地，我们可以只使用一个字符串存储每个状态的字符串，在递归回溯的过程中，需要将路径中的当前节点移除，以回到上一个节点。

具体实现方法如下：

1. 定义一个结果数组和一个路径数组。
2. 从根节点开始递归，递归函数的参数为当前节点、结果数组和路径数组。
    a. 如果当前节点为空，返回。
    b. 将当前节点的值加入到路径数组中。
    c. 如果当前节点为叶子节点，将路径数组中的所有元素拼接成字符串，并将该字符串存储到结果数组中。
    d. 递归遍历当前节点的左子树。
    e. 递归遍历当前节点的右子树。
    f. 回溯，将路径数组中的最后一个元素移除，以返回到上一个节点。
3. 返回结果数组。

C++ 算法代码：

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution
{
public:
    vector<string> ret; // 记录结果
    vector<string> binaryTreePaths(TreeNode* root)
    {
        string path;
        if(root == nullptr) return ret;
        dfs(root, path);
        return ret;
    }

    void dfs(TreeNode* root, string path)
    {
        path += to_string(root->val);
        if(root->left == nullptr && root->right == nullptr)
        {
            ret.push_back(path);
            return;
        }

        path += "->";
        if(root->left) dfs(root->left, path);
        if(root->right) dfs(root->right, path);
    }
};
```


C++ 运行结果：

Java 算法代码：

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution
{
    List<String> ret;
    public List<String> binaryTreePaths(TreeNode root)
    {
        ret = new ArrayList<>();
        dfs(root, new StringBuffer());
        return ret;
    }

    void dfs(TreeNode root, StringBuffer _path)
    {
        StringBuffer path = new StringBuffer(_path);
        path.append(Integer.toString(root.val));
        if(root.left == null && root.right == null)
        {
            ret.add(path.toString());
            return;
        }
        path.append("->");
        if(root.left != null) dfs(root.left, path);

        if(root.right != null) dfs(root.right, path);
    }
}
```

Java 运行结果：

穷举vs暴搜vs深搜vs回溯vs剪枝

什么是回溯算法

回溯算法是一种经典的递归算法，通常用于解决组合问题、排列问题和搜索问题等。

回溯算法的基本思想：从一个初始状态开始，按照一定的规则向前搜索，当搜索到某个状态无法前进时，回退到前一个状态，再按照其他的规则搜索。回溯算法在搜索过程中维护一个状态树，通过遍历状态树来实现对所有可能解的搜索。

回溯算法的核心思想：“试错”，即在搜索过程中不断地做出选择，如果选择正确，则继续向前搜索；否则，回退到上一个状态，重新做出选择。回溯算法通常用于解决具有多个解，且每个解都需要搜索才能找到的问题。

回溯算法的模板

```
void backtrack(vector<int>& path, vector<int>& choice, ...) {
    // 满足结束条件
    if (/* 满足结束条件 */) {
        // 将路径添加到结果集中
        res.push_back(path);
        return;
    }

    // 遍历所有选择
    for (int i = 0; i < choices.size(); i++) {
        // 做出选择
        path.push_back(choices[i]);
        // 做出当前选择后继续搜索
        backtrack(path, choices);
        // 撤销选择
        path.pop_back();
    }
}
```

其中，`path` 表示当前已经做出的选择，`choices` 表示当前可以做的选择。在回溯算法中，我们需要做出选择，然后递归地调用回溯函数。如果满足结束条件，则将当前路径添加到结果集中；否则，我们需要撤销选择，回到上一个状态，然后继续搜索其他的选择。

回溯算法的时间复杂度通常较高，因为它需要遍历所有可能的解。但是，回溯算法的空间复杂度较低，因为它只需要维护一个状态树。在实际应用中，回溯算法通常需要通过剪枝等方法进行优化，以减少搜索的次数，从而提高算法的效率。

回溯算法的应用

组合问题

组合问题是指从给定的一组数（不重复）中选取出所有可能的 k 个数的组合。例如，给定数集 [1,2,3]，要求选取 k=2 个数的所有组合。

结果为：

```
[1,2]
[1,3]
[2,3]
```

排列问题

排列问题是指从给定的一组数（不重复）中选取出所有可能的 k 个数的排列。例如，给定数集 [1,2,3]，要求选取 k=2 个数的所有排列。

结果为：

```
[1,2]
[2,1]
[1,3]
[3,1]
[2,3]
[3,2]
```

子集问题

子集问题是指从给定的一组数中选取出所有可能的子集，其中每个子集中的元素可以按照任意顺序排列。例如，给定数集 [1,2,3]，要求选取所有可能的子集。

结果为：

```
[]
```

```
[1]
[2]
[3]
[1,2]
[1,3]
[2,3]
[1,2,3]
```

总结

回溯算法是一种非常重要的算法，可以解决许多组合问题、排列问题和搜索问题等。回溯算法的核心思想是搜索状态树，通过遍历状态树来实现对所有可能解的搜索。回溯算法的模板非常简单，但是实现起来需要注意一些细节，比如如何做出选择、如何撤销选择等。

12. 全排列（medium）

1. **题目链接：46. 全排列**
2. **题目描述：**

> 给定一个不含重复数字的数组 nums ，返回其 所有可能的全排列 。你可以 按任意顺序 返回答案。

- 示例 1：

> 输入：nums = [1,2,3]
>
> 输出：[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

- 示例 2：

> 输入：nums = [0,1]
>
> 输出：[[0,1],[1,0]]

- 示例 3：

> 输入：nums = [1]
>
> 输出：[[1]]

- 提示：

> 1 <= nums.length <= 6
>
> -10 <= nums[i] <= 10
>
> nums 中的所有整数 互不相同

3. **解法：**

算法思路：

典型的回溯题目，我们需要在每一个位置上考虑所有的可能情况并且不能出现重复。通过深度优先搜索的方式，不断地枚举每个数在当前位置的可能性，并回溯到上一个状态，直到枚举完所有可能性，得到正确的结果。

每个数是否可以放入当前位置，只需要判断这个数在之前是否出现即可。具体地，在这道题目中，我们可以通过一个递归函数 backtrack 和标记数组 visited 来实现全排列。

递归函数设计：void backtrack(vector<vector<int>>& res, vector<int>& nums, vector<bool>& visited, vector<int>& ans, int step, int len)

参数：step（当前需要填入的位置），len（数组长度）；

返回值：无；

函数作用：查找所有合理的排列并存储在答案列表中。

递归流程如下：

1. 首先定义一个二维数组 res 用来存放所有可能的排列，一个一维数组 ans 用来存放每个状态的排列，一个一维数组 visited 标记元素，然后从第一个位置开始进行递归；
2. 在每个递归的状态中，我们维护一个步数 step，表示当前已经处理了几个数字；
3. 递归结束条件：当 step 等于 nums 数组的长度时，说明我们已经处理完了所有数字，将当前数组存入结果中；
4. 在每个递归状态中，枚举所有下标 i，若这个下标未被标记，则使用 nums 数组中当前下标的元素：
    a. 将 visited[i] 标记为 1；
    b. ans 数组中第 step 个元素被 nums[i] 覆盖；
    c. 对第 step+1 个位置进行递归；
    d. 将 visited[i] 重新赋值为 0，表示回溯；
5. 最后，返回 res。

- 特别地，我们可以不使用标记数组，直接遍历 step 之后的元素（未被使用），然后将其与需要递归的位置进行交换即可。

C++ 算法代码：

```
1 class Solution
2 {
3     vector<vector<int>> ret;
4     vector<int> path;
```

```
    bool check[7];

public:
    vector<vector<int>> permute(vector<int>& nums)
    {
        dfs(nums);
        return ret;
    }

    void dfs(vector<int>& nums)
    {
        if(path.size() == nums.size())
        {
            ret.push_back(path);
            return;
        }

        for(int i = 0; i < nums.size(); i++)
        {
            if(!check[i])
            {
                path.push_back(nums[i]);
                check[i] = true;
                dfs(nums);
                // 回溯 -> 恢复现场
                path.pop_back();
                check[i] = false;
            }
        }
    }
};
```

C++ 运行结果：

Java 算法代码：

```
class Solution
{
```

```java
    List<List<Integer>> ret;
    List<Integer> path;
    boolean[] check;

    public List<List<Integer>> permute(int[] nums)
    {
        ret = new ArrayList<>();
        path = new ArrayList<>();
        check = new boolean[nums.length];

        dfs(nums);
        return ret;
    }

    public void dfs(int[] nums)
    {
        if(nums.length == path.size())
        {
            ret.add(new ArrayList<>(path));
            return;
        }

        for(int i = 0; i < nums.length; i++)
        {
            if(check[i] == false)
            {
                path.add(nums[i]);
                check[i] = true;
                dfs(nums);
                // 回溯 -> 恢复现场
                check[i] = false;
                path.remove(path.size() - 1);
            }
        }
    }
}
```


Java 运行结果：

13. 子集（medium）

1. **题目链接：78. 子集**

2. **题目描述：**

> 给你一个整数数组 nums ，数组中的元素 互不相同 。返回该数组所有可能的子集（幂集）。
>
> 解集 不能 包含重复的子集。你可以按 任意顺序 返回解集。

- 示例 1：

> 输入：nums = [1,2,3]
>
> 输出：[[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]

- 示例 2：

> 输入：nums = [0]
>
> 输出：[[],[0]]

- 提示：

> 1 <= nums.length <= 10
>
> -10 <= nums[i] <= 10
>
> nums 中的所有元素 互不相同

3. **解法：**

算法思路：

为了获得 nums 数组的所有子集，我们需要对数组中的每个元素进行选择或不选择的操作，即 nums 数组一定存在 2^(数组长度) 个子集。对于查找子集，具体可以定义一个数组，来记录当前的状态，并对其进行递归。

对于每个元素有两种选择：1. 不进行任何操作；2. 将其添加至当前状态的集合。在递归时我们需要保证递归结束时当前的状态与进行递归操作前的状态不变，而当我们在选择进行步骤 2 进行递归时，当前状态会发生变化，因此我们需要在递归结束时撤回添加操作，即进行回溯。

递归函数设计：void dfs(vector<vector<int>>& res, vector<int>& ans, vector<int>& nums, int step)

参数：step（当前需要处理的元素下标）；

返回值：无；

函数作用：查找集合的所有子集并存储在答案列表中。

递归流程如下：

1. 递归结束条件：如果当前需要处理的元素下标越界，则记录当前状态并直接返回；
2. 在递归过程中，对于每个元素，我们有两种选择：
    - 不选择当前元素，直接递归到下一个元素；
    - 选择当前元素，将其添加到数组末尾后递归到下一个元素，然后在递归结束时撤回添加操作；
3. 所有符合条件的状态都被记录下来，返回即可。

C++ 算法代码：

```cpp
// 解法一:
class Solution
{
    vector<vector<int>> ret;
    vector<int> path;
public:
    vector<vector<int>> subsets(vector<int>& nums)
    {
        dfs(nums, 0);
        return ret;
    }

    void dfs(vector<int>& nums, int pos)
    {
        if(pos == nums.size())
        {
            ret.push_back(path);
            return;
        }

        // 选
        path.push_back(nums[pos]);
        dfs(nums, pos + 1);
        path.pop_back(); // 恢复现场

        // 不选
        dfs(nums, pos + 1);
    }
};

// 解法二:
class Solution
{
    vector<vector<int>> ret;
    vector<int> path;
```

```cpp
public:
    vector<vector<int>> subsets(vector<int>& nums)
    {

        dfs(nums, 0);
        return ret;
    }

    void dfs(vector<int>& nums, int pos)
    {
        ret.push_back(path);
        for(int i = pos; i < nums.size(); i++)
        {
            path.push_back(nums[i]);
            dfs(nums, i + 1);
            path.pop_back(); // 恢复现场
        }
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
// 解法一：
class Solution
{
    List<List<Integer>> ret;
    List<Integer> path;

    public List<List<Integer>> subsets(int[] nums)
    {
        ret = new ArrayList<>();
        path = new ArrayList<>();

        dfs(nums, 0);
        return ret;
    }
```

```java

    public void dfs(int[] nums, int pos)
    {
        if(pos == nums.length)
        {
            ret.add(new ArrayList<>(path));
            return;
        }

        // 选
        path.add(nums[pos]);
        dfs(nums, pos + 1);
        path.remove(path.size() - 1); // 恢复现场

        // 不选
        dfs(nums, pos + 1);
    }
}

// 解法二:
class Solution
{
    List<List<Integer>> ret;
    List<Integer> path;

    public List<List<Integer>> subsets(int[] nums)
    {
        ret = new ArrayList<>();
        path = new ArrayList<>();

        dfs(nums, 0);
        return ret;
    }

    public void dfs(int[] nums, int pos)
    {
        ret.add(new ArrayList<>(path));

        for(int i = pos; i < nums.length; i++)
        {
            path.add(nums[i]);
            dfs(nums, i + 1);
            path.remove(path.size() - 1); // 恢复现场
        }
    }
}
```

Java 运行结果：

综合练习

14. 找出所有子集的异或总和再求和（easy）

1. **题目链接：1863. 找出所有子集的异或总和再求和**
2. **题目描述：**

> 一个数组的 异或总和 定义为数组中所有元素按位 XOR 的结果；如果数组为 空 ，则异或总和为 0 。
>
> 例如，数组 [2,5,6] 的 异或总和 为 2 XOR 5 XOR 6 = 1 。
>
> 给你一个数组 nums ，请你求出 nums 中每个 子集 的 异或总和 ，计算并返回这些值相加之 和 。
>
> 注意：在本题中，元素 相同 的不同子集应 多次 计数。
>
> 数组 a 是数组 b 的一个 子集 的前提条件是：从 b 删除几个（也可能不删除）元素能够得到 a 。

- 示例 1：

> 输入：nums = [1,3]
>
> 输出：6
>
> 解释：[1,3] 共有 4 个子集：
>
> - 空子集的异或总和是 0 。
> - [1] 的异或总和为 1 。
> - [3] 的异或总和为 3 。
> - [1,3] 的异或总和为 1 XOR 3 = 2 。
>
> 0 + 1 + 3 + 2 = 6

- 示例 2：

> 输入：nums = [5,1,6]
>
> 输出：28
>
> 解释：[5,1,6] 共有 8 个子集：
>
> - 空子集的异或总和是 0 。

- [5] 的异或总和为 5 。
- [1] 的异或总和为 1 。
- [6] 的异或总和为 6 。
- [5,1] 的异或总和为 5 XOR 1 = 4 。
- [5,6] 的异或总和为 5 XOR 6 = 3 。
- [1,6] 的异或总和为 1 XOR 6 = 7 。
- [5,1,6] 的异或总和为 5 XOR 1 XOR 6 = 2 。

0 + 5 + 1 + 6 + 4 + 3 + 7 + 2 = 28

- 示例 3：

输入：nums = [3,4,5,6,7,8]

输出：480

解释：每个子集的全部异或总和值之和为 480 。

- 提示：

1 <= nums.length <= 12

1 <= nums[i] <= 20

3. **解法（递归）：**

算法思路：

所有子集可以解释为：每个元素选择在或不在一个集合中（因此，子集有 2^n 个）。本题我们需要求出所有子集，将它们的异或和相加。因为异或操作满足交换律，所以我们可以定义一个变量，直接记录当前状态的异或和。使用递归保存当前集合的状态（异或和），选择将当前元素添加至当前状态与否，并依次递归数组中下一个元素。当递归到空元素时，表示所有元素都被考虑到，记录当前状态（将当前状态的异或和添加至答案中）。

例如集合中的元素为 [1, 2]，则它的子集状态选择过程如下：

```
1          []
2       /      \
3     []        [1]        //第一个元素选择与否
4    /  \       /  \
5   []  [2]   [1] [1, 2]   //第二个元素选择与否，每个状态到这一层时需要记录异或和
```

递归函数设计：void dfs(int val, int idx, vector<int>& nums)

参数：val（当前状态的异或和），idx（当前需要处理的元素下标，处理过程：选择将其添加至当前状态或不进行操作）；

返回值：无；

函数作用：选择对元素进行添加与否处理。

递归流程：

1. 递归结束条件：当前下标与数组长度相等，即已经越界，表示已经考虑到所有元素；
   a. 将当前异或和添加至答案中，并返回；
2. 考虑将当前元素添加至当前状态，当前状态更新为与当前元素值的异或和，然后递归下一个元素；
3. 考虑不选择当前元素，当前状态不变，直接递归下一个元素；

C++ 算法代码：

```cpp
class Solution
{
    int path;
    int sum;

public:
    int subsetXORSum(vector<int>& nums)
    {
        dfs(nums, 0);
        return sum;
    }

    void dfs(vector<int>& nums, int pos)
    {
        sum += path;
        for(int i = pos; i < nums.size(); i++)
        {
            path ^= nums[i];
            dfs(nums, i + 1);
            path ^= nums[i]; // 恢复现场
        }
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution
{
    int path;
    int sum;

    public int subsetXORSum(int[] nums)
    {
        dfs(nums, 0);
        return sum;
    }

    public void dfs(int[] nums, int pos)
    {
        sum += path;
        for(int i = pos; i < nums.length; i++)
        {
            path ^= nums[i];
            dfs(nums, i + 1);
            path ^= nums[i]; // 恢复现场
        }
    }
}
```

Java 运行结果：

15. 全排列 II （medium）

1. **题目链接：47. 全排列 II**
2. **题目描述：**

> 给定一个可包含重复数字的序列 nums ，按任意顺序 返回所有不重复的全排列。

- 示例 1：

> 输入：nums = [1,1,2]
>
> 输出：
>
> [[1,1,2],
>
> [1,2,1],
>
> [2,1,1]]

- 示例 2：

> 输入：nums = [1,2,3]
>
> 输出：[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

- 提示：

> 1 <= nums.length <= 8
>
> -10 <= nums[i] <= 10

3. **解法：**

算法思路：

因为题目不要求返回的排列顺序，因此我们可以对初始状态排序，将所有相同的元素放在各自相邻的位置，方便之后操作。因为重复元素的存在，我们在选择元素进行全排列时，可能会存在重复排列，例如：[1, 2, 1]，所有的 下标排列 为：

```
123
132
213
231
312
321
```

按照以上下标进行排列的结果为：

```
121
112
211
211
112
121
```

可以看到，有效排列只有三种[1, 1, 2]，[1, 2, 1]，[2, 1, 1]，其中每个排列都出现两次。因此，我们需要对相同元素定义一种规则，使得其组成的排列不会形成重复的情况：

1. 我们可以将相同的元素按照排序后的下标顺序出现在排列中，通俗来讲，若元素 s 出现 x 次，则排序后的第 2 个元素 s 一定出现在第 1 个元素 s 后面，排序后的第 3 个元素 s 一定出现在第 2 个元素 s 后面，以此类推，此时的全排列一定不会出现重复结果。
2. 例如：a1=1，a2=1，a3=2，排列结果为 [1, 1, 2] 的情况只有一次，即 a1 在 a2 前面，因为 a2 不会出现在 a1 前面从而避免了重复排列。
3. 我们在每一个位置上考虑所有的可能情况并且不出现重复；
4. *注意*：若当前元素的前一个相同元素未出现在当前状态中，则当前元素也不能直接放入当前状态的数组，此做法可以保证相同元素的排列顺序与排序后的相同元素的顺序相同，即避免了重复排列出现。
5. 通过深度优先搜索的方式，不断地枚举每个数在当前位置的可能性，并在递归结束时回溯到上一个状态，直到枚举完所有可能性，得到正确的结果。

递归函数设计：void backtrack(vector<int>& nums, int idx)

参数：idx（当前需要填入的位置）；

返回值：无；

函数作用：查找所有合理的排列并存储在答案列表中。

递归流程如下：

1. 定义一个二维数组 ans 用来存放所有可能的排列，一个一维数组 perm 用来存放每个状态的排列，一个一维数组 visited 标记元素，然后从第一个位置开始进行递归；
2. 在每个递归的状态中，我们维护一个步数 idx，表示当前已经处理了几个数字；
3. 递归结束条件：当 idx 等于 nums 数组的长度时，说明我们已经处理完了所有数字，将当前数组存入结果中；
4. 在每个递归状态中，枚举所有下标 i，若这个下标未被标记，并且在它之前的相同元素被标记过，则使用 nums 数组中当前下标的元素：
    a. 将 visited[i] 标记为 1；
    b. 将 nums[i] 添加至 perm 数组末尾；
    c. 对第 step+1 个位置进行递归；
    d. 将 visited[i] 重新赋值为 0，并删除 perm 末尾元素表示回溯；
5. 最后，返回 ans。

C++ 算法代码：

```cpp
class Solution
{
    vector<int> path;
    vector<vector<int>> ret;
    bool check[9];

public:
    vector<vector<int>> permuteUnique(vector<int>& nums)
    {
        sort(nums.begin(), nums.end());

        dfs(nums, 0);
        return ret;
    }

    void dfs(vector<int>& nums, int pos)
    {
        if(pos == nums.size())
        {
            ret.push_back(path);
            return;
        }

        for(int i = 0; i < nums.size(); i++)
        {
            // 剪枝
            if(check[i] == false && (i == 0 || nums[i] != nums[i - 1] || check[i - 1] != false))
            {
                path.push_back(nums[i]);
                check[i] = true;
                dfs(nums, pos + 1);
                path.pop_back(); // 恢复现场
                check[i] = false;
            }

        }
    }
};
```

C++ 运行结果：

C++

时间 4 ms　击败 94.4%　内存 8.7 MB　击败 71.73%

Java 算法代码：

```java
class Solution
{
    List<Integer> path;
    List<List<Integer>> ret;
    boolean[] check;

    public List<List<Integer>> permuteUnique(int[] nums)
    {
        path = new ArrayList<>();
        ret = new ArrayList<>();
        check = new boolean[nums.length];

        Arrays.sort(nums);
        dfs(nums, 0);
        return ret;
    }

    public void dfs(int[] nums, int pos)
    {
        if(pos == nums.length)
        {
            ret.add(new ArrayList<>(path));
            return;
        }

        for(int i = 0; i < nums.length; i++)
        {
            // 剪枝
            if(check[i] == false && (i == 0 || nums[i] != nums[i - 1] || check[i - 1] != false))
            {
                path.add(nums[i]);
                check[i] = true;
                dfs(nums, pos + 1);
                // 回溯 -> 恢复现场
                path.remove(path.size() - 1);
                check[i] = false;
```

```
            }
        }
    }
}
```

Java 运行结果：

16. 电话号码的字母组合（medium）

1. **题目链接：17. 电话号码的字母组合**

2. **题目描述：**

> 给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。答案可以按 任意顺序 返回。
>
> 给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。

- 示例 1：

> 输入：digits = "23"
>
> 输出：["ad","ae","af","bd","be","bf","cd","ce","cf"]

- 示例 2：

> 输入：digits = ""
>
> 输出：[]

- 示例 3：

> 输入：digits = "2"
>
> 输出：["a","b","c"]

- 提示：

> 0 <= digits.length <= 4
>
> digits[i] 是范围 ['2', '9'] 的一个数字。

3. **解法：**

算法思路：

每个位置可选择的字符与其他位置并不冲突，因此不需要标记已经出现的字符，只需要将每个数字对应的字符依次填入字符串中进行递归，在回溯是撤销填入操作即可。

- 在递归之前我们需要定义一个字典 phoneMap，记录 2~9 各自对应的字符。

递归函数设计：void backtrack(unordered_map<char, string>& phoneMap, string& digits, int index)

参数：index （已经处理的元素个数），ans （字符串当前状态），res （所有成立的字符串）；

返回值：无

函数作用：查找所有合理的字母组合并存储在答案列表中。

递归函数流程如下：

1. 递归结束条件：当 index 等于 digits 的长度时，将 ans 加入到 res 中并返回；
2. 取出当前处理的数字 digit，根据 phoneMap 取出对应的字母列表 letters；
3. 遍历字母列表 letters，将当前字母加入到组合字符串 ans 的末尾，然后递归处理下一个数字（传入 index + 1，表示处理下一个数字）；
4. 递归处理结束后，将加入的字母从 ans 的末尾删除，表示回溯。
5. 最终返回 res 即可。

C++ 算法代码：

```
class Solution
{
    string hash[10] = {"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs",
"tuv", "wxyz"};
    string path;
    vector<string> ret;

public:
    vector<string> letterCombinations(string digits)
    {
        if(digits.size() == 0) return ret;

        dfs(digits, 0);
        return ret;
    }

    void dfs(string& digits, int pos)
    {
        if(pos == digits.size())
        {
```

```
            ret.push_back(path);
            return;
        }

        for(auto ch : hash[digits[pos] - '0'])
        {
            path.push_back(ch);
            dfs(digits, pos + 1);
            path.pop_back(); // 恢复现场
        }
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution
{
    String[] hash = {"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};
    List<String> ret;
    StringBuffer path;

    public List<String> letterCombinations(String digits)
    {
        ret = new ArrayList<>();
        path = new StringBuffer();

        if(digits.length() == 0) return ret;

        dfs(digits, 0);
        return ret;
    }

    public void dfs(String digits, int pos)
    {
        if(pos == digits.length())
```

```
21          {
22              ret.add(path.toString());
23              return;
24          }
25
26          String cur = hash[digits.charAt(pos) - '0'];
27          for(int i = 0; i < cur.length(); i++)
28          {
29              path.append(cur.charAt(i));
30              dfs(digits, pos + 1);
31              path.deleteCharAt(path.length() - 1); // 恢复现场
32          }
33      }
34  }
```

Java 运行结果：

17. 括号生成（medium）

1. **题目链接：22. 括号生成**
2. **题目描述：**

> 括号。设计一种算法，打印n对括号的所有合法的（例如，开闭一一对应）组合。
>
> 说明：解集不能包含重复的子集。

- 例如：

> 给出 n = 3，生成结果为：
>
> [
>
> "((()))",
>
> "(()())",
>
> "(())()",
>
> "()(())",
>
> "()()()"
>
>]

3. **解法：**

算法思路：

从左往右进行递归，在每个位置判断放置左右括号的可能性，若此时放置左括号合理，则放置左括号继续进行递归，右括号同理。

一种判断括号是否合法的方法：从左往右遍历，左括号的数量始终大于等于右括号的数量，并且左括号的**总数量**与右括号的总数量相等。因此我们在递归时需要进行以下判断：

1. 放入左括号时需判断此时左括号数量是否小于字符串总长度的一半（若左括号的数量大于等于字符串长度的一半时继续放置左括号，则左括号的总数量一定大于右括号的总数量）；
2. 放入右括号时需判断此时右括号数量是否小于左括号数量。

递归函数设计：void dfs(int step, int left)

参数：step（当前需要填入的位置），left（当前状态的字符串中的左括号数量）；

返回值：无；

函数作用：查找所有合理的括号序列并存储在答案列表中。

递归函数参数设置为当前状态的字符串长度以及当前状态的左括号数量，**递归流程如下：**

1. 递归结束条件：当前状态字符串长度与 2*n 相等，记录当前状态并返回；
2. 若此时左括号数量小于字符串总长度的一半，则在当前状态的字符串末尾添加左括号并继续递归，递归结束撤销添加操作；
3. 若此时右括号数量小于左括号数量（右括号数量可以由当前状态的字符串长度减去左括号数量求得），则在当前状态的字符串末尾添加右括号并递归，递归结束撤销添加操作；

C++ 算法代码：

```cpp
class Solution
{
    int left, right, n;
    string path;
    vector<string> ret;

public:
    vector<string> generateParenthesis(int _n)
    {
        n = _n;
        dfs();
        return ret;
```

```
    }

    void dfs()
    {
        if(right == n)
        {
            ret.push_back(path);
            return;
        }

        if(left < n) // 添加左括号
        {
            path.push_back('('); left++;
            dfs();
            path.pop_back(); left--; // 恢复现场
        }

        if(right < left) // 添加右括号
        {
            path.push_back(')'); right++;
            dfs();
            path.pop_back(); right--; // 恢复现场
        }
    }
};
```

C++ 运行结果：

Java 算法代码：

```
class Solution
{
    int left, right, n;
    StringBuffer path;
    List<String> ret;

    public List<String> generateParenthesis(int _n)
    {
```

```
 9          n = _n;
10          path = new StringBuffer();
11          ret = new ArrayList<>();
12
13          dfs();
14          return ret;
15      }
16
17      public void dfs()
18      {
19          if(right == n)
20          {
21              ret.add(path.toString());
22              return;
23          }
24
25          if(left < n) // 添加左括号
26          {
27              path.append('('); left++;
28              dfs();
29              path.deleteCharAt(path.length() - 1); left--; // 恢复现场
30          }
31
32          if(right < left) // 添加收右括号
33          {
34              path.append(')'); right++;
35              dfs();
36              path.deleteCharAt(path.length() - 1); right--; // 恢复现场
37          }
38      }
39  }
```

Java 运行结果：

18. 组合（medium）

1. **题目链接：77. 组合**

2. **题目描述：**

> 给定两个整数 n 和 k，返回范围 [1, n] 中所有可能的 k 个数的组合。
>
> 你可以按 任何顺序 返回答案。

- 示例 1：

> 输入：n = 4, k = 2
>
> 输出：
>
> [
>
> [2,4],
>
> [3,4],
>
> [2,3],
>
> [1,2],
>
> [1,3],
>
> [1,4],
>
>]

- 示例 2：

> 输入：n = 1, k = 1
>
> 输出：[[1]]

- 提示：

> 1 <= n <= 20
>
> 1 <= k <= n

3. **解法（回溯）：**

算法思路：

题目要求我们从 1 到 n 中选择 k 个数的所有组合，其中不考虑顺序。也就是说，[1,2] 和 [2,1] 等价。我们需要找出所有的组合，但不能重复计算相同元素的不同顺序的组合。对于选择组合，我们需要进行如下流程：

1. 所有元素分别作为首位元素进行处理；
2. 在之后的位置上同理，选择所有元素分别作为当前位置元素进行处理；
3. 为避免计算重复组合，规定选择之后位置的元素时必须比前一个元素大，这样就不会有重复的组合（[1,2] 和 [2,1] 中 [2,1] 不会出现）。

递归函数设计：void dfs(vector<vector<int>>& ans, vector<int>& v, int step, int &n, int &k)

参数：step（当前需要进行处理的位置）；

返回值：无；

函数作用：某个元素作为首位元素出现时，查找所有可能的组合。

具体**实现方法如下：**

1. 定义一个二维数组和一维数组。二维数组用来记录所有组合，一维数组用来记录当前状态下的组合。
2. 遍历 1 到 n-k+1，以当前数作为组合的首位元素进行递归（从 n-k+1 到 n 作为首位元素时，组合中一定不会存在 k 个元素）。
3. 递归函数的参数为两个数组、当前步骤以及 n 和 k。递归流程如下：
   a. 结束条件：当前组合中已经有 k 个元素，将当前组合存进二维数组并返回。
      - 剪枝：如果当前位置之后的所有元素放入组合也不能满足组合中存在 k 个元素，直接返回。
   b. 从当前位置的下一个元素开始遍历到 n，将元素赋值到当前位置，递归下一个位置。

C++ 算法代码：

```cpp
class Solution
{
    vector<int> path;
    vector<vector<int>> ret;
    int n, k;

public:
    vector<vector<int>> combine(int _n, int _k)
    {
        n = _n; k = _k;
        dfs(1);
        return ret;
    }

    void dfs(int start)
    {
        if(path.size() == k)
        {
            ret.push_back(path);
            return;
        }

        for(int i = start; i <= n; i++)
        {
            path.push_back(i);
```

```
            dfs(i + 1);
            path.pop_back(); // 恢复现场
        }
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution
{
    List<Integer> path;
    List<List<Integer>> ret;
    int n, k;

    public List<List<Integer>> combine(int _n, int _k)
    {
        n = _n; k = _k;
        path = new ArrayList<>();
        ret = new ArrayList<>();

        dfs(1);
        return ret;
    }

    public void dfs(int start)
    {
        if(path.size() == k)
        {
            ret.add(new ArrayList<>(path));
            return;
        }

        for(int i = start; i <= n; i++)
        {
            path.add(i);
            dfs(i + 1);
```

```
29                 path.remove(path.size() - 1); // 恢复现场
30             }
31         }
32     }
```

Java 运行结果：

19. 目标和（medium）

1. **题目链接：494. 目标和**

2. **题目描述：**

> 给你一个整数数组 nums 和一个整数 target。
>
> 向数组中的每个整数前添加 '+' 或 '-' ，然后串联起所有整数，可以构造一个 表达式 ：
>
> 例如，nums = [2, 1] ，可以在 2 之前添加 '+' ，在 1 之前添加 '-' ，然后串联起来得到表达式 "+2-1" 。
>
> 返回可以通过上述方法构造的、运算结果等于 target 的不同 表达式 的数目。

- 示例 1：

> 输入：nums = [1,1,1,1,1], target = 3
>
> 输出：5
>
> 解释：
>
> 一共有 5 种方法让最终目标和为 3 。
>
> -1 + 1 + 1 + 1 + 1 = 3
>
> +1 - 1 + 1 + 1 + 1 = 3
>
> +1 + 1 - 1 + 1 + 1 = 3
>
> +1 + 1 + 1 - 1 + 1 = 3
>
> +1 + 1 + 1 + 1 - 1 = 3

- 示例 2：

> 输入：nums = [1], target = 1
>
> 输出：1

3. **解法（回溯）：**

算法思路：

对于每个数，可以选择加上或减去它，依次枚举每一个数字，在每个数都被选择时检查得到的和是否等于目标值。如果等于，则记录结果。

需要注意的是，为了优化时间复杂度，可以提前计算出数组中所有数字的和 sum，以及数组的长度 len。这样可以快速判断当前的和减去剩余的所有数是否已经超过了目标值 target ，或者当前的和加上剩下的数的和是否小于目标值 target，如果满足条件，则可以直接回溯。

递归流程：

1. 递归结束条件：index 与数组长度相等，判断当前状态的 sum 是否与目标值相等，若是计数加一；
2. 选择当前元素进行加操作，递归下一个位置，并更新参数 sum；
3. 选择当前元素进行减操作，递归下一个位置，并更新参数 sum；

- 特别地，此问题可以转化为另一个问题：若所有元素初始状态均为减，选择其中几个元素将他们的状态修改为加，计算修改后的元素和与目标值相等的方案个数。

1. 选择其中 x 个元素进行修改，并且这 x 个元素的和为 y；
2. 检查使得 -sum+2*y=target（移项：y=(sum+target)/2）成立的方案个数，即选择 x 个元素和为 (sum+target)/2 的方案个数；
   a. 若 sum+target 为奇数，则不存在这种方案；
3. 递归流程：
   a. 传入参数：index（当前要处理的元素下标），sum（当前状态和），nums（元素数组），aim（目标值：(sum+target)/2）；
   b. 递归结束条件：index 与数组长度相等，判断当前 sum 是否与目标值相等，若是返回 1，否则返回 0；
   c. 返回 递归选择当前元素 以及 递归不选择当前元素 函数值的和。

C++ 算法代码：

```
class Solution
{
    int ret, aim;

public:
    int findTargetSumWays(vector<int>& nums, int target)
    {
        aim = target;
```

```
        dfs(nums, 0, 0);
        return ret;
    }

    void dfs(vector<int>& nums, int pos, int path)
    {
        if(pos == nums.size())
        {
            if(path == aim) ret++;
            return;
        }

        // 加法
        dfs(nums, pos + 1, path + nums[pos]);

        // 减法
        dfs(nums, pos + 1, path - nums[pos]);
    }
};
```

C++ 运行结果：

Java 算法代码：

```
class Solution
{
    int ret, aim;

    public int findTargetSumWays(int[] nums, int target)
    {
        aim = target;
        dfs(nums, 0, 0);
        return ret;
    }

    public void dfs(int[] nums, int pos, int path)
    {
        if(pos == nums.length)
```

```
15          {
16              if(path == aim) ret++;
17              return;
18          }
19
20          // 加法
21          dfs(nums, pos + 1, path + nums[pos]);
22
23          // 减法
24          dfs(nums, pos + 1, path - nums[pos]);
25      }
26  }
```

Java 运行结果：

20. 组合总和（medium）

1. **题目链接：39. 组合总和**
2. **题目描述：**

> 给你一个 无重复元素 的整数数组 candidates 和一个目标整数 target ，找出 candidates 中可以使数字和为目标数 target 的 所有 不同组合 ，并以列表形式返回。你可以按 任意顺序 返回这些组合。
>
> candidates 中的 同一个 数字可以 无限制重复被选取 。如果至少一个数字的被选数量不同，则两种组合是不同的。
>
> 对于给定的输入，保证和为 target 的不同组合数少于 150 个。

- 示例 1：

> 输入：candidates = [2,3,6,7], target = 7
>
> 输出：[[2,2,3],[7]]
>
> 解释：
>
> 2 和 3 可以形成一组候选，2 + 2 + 3 = 7 。注意 2 可以使用多次。
>
> 7 也是一个候选， 7 = 7 。
>
> 仅有这两种组合。

- 示例 2：

> 输入: candidates = [2,3,5], target = 8
>
> 输出: [[2,2,2,2],[2,3,3],[3,5]]

- 示例 3：

> 输入: candidates = [2], target = 1
>
> 输出: []

- 提示：

> 1 <= candidates.length <= 30
>
> 2 <= candidates[i] <= 40
>
> candidates 的所有元素 互不相同
>
> 1 <= target <= 40

3. **解法：**

算法思路：

candidates 的所有元素 互不相同，因此我们在递归状态时只需要对每个元素进行如下判断：

1. 跳过，对下一个元素进行判断；
2. 将其添加至当前状态中，我们在选择添加当前元素时，之后仍可以继续选择当前元素（可以重复选择同一元素）。

- 因此，我们在选择当前元素并向下传递下标时，应该直接传递当前元素下标。

递归函数设计：void dfs(vector<int>& candidates, int target, vector<vector<int>>& ans, vector<int>& combine, int idx)

参数：target（当前状态和与目标值的差），idx（当前需要处理的元素下标）；

返回值：无；

函数作用：向下传递两个状态（跳过或者选择当前元素），找出所有组合使得元素和为目标值。

递归函数流程如下：

1. 结束条件：
   a. 当前需要处理的元素下标越界；
   b. 当前状态的元素和已经与目标值相同；
2. 跳过当前元素，当前状态不变，对下一个元素进行处理；
3. 选择将当前元素添加至当前状态，并保留状态继续对当前元素进行处理，递归结束时撤销添加操作。

C++ 算法代码：

```cpp
class Solution
{
    int aim;
    vector<int> path;
    vector<vector<int>> ret;

public:
    vector<vector<int>> combinationSum(vector<int>& nums, int target)
    {
        aim = target;
        dfs(nums, 0, 0);
        return ret;
    }

    void dfs(vector<int>& nums, int pos, int sum)
    {
        if(sum == aim)
        {
            ret.push_back(path);
            return;
        }
        if(sum > aim || pos == nums.size()) return;

        // 枚举个数
        for(int k = 0; k * nums[pos] + sum <= aim; k++)
        {
            if(k) path.push_back(nums[pos]);
            dfs(nums, pos + 1, sum + k * nums[pos]);
        }

        // 恢复现场
        for(int k = 1; k * nums[pos] + sum <= aim; k++)
        {
            path.pop_back();
        }
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution
{
    int aim;
    List<Integer> path;
    List<List<Integer>> ret;

    public List<List<Integer>> combinationSum(int[] nums, int target)
    {
        path = new ArrayList<>();
        ret = new ArrayList<>();
        aim = target;

        dfs(nums, 0, 0);
        return ret;
    }

    public void dfs(int[] nums, int pos, int sum)
    {
        if(sum == aim)
        {
            ret.add(new ArrayList<>(path));
            return;
        }
        if(sum > aim || pos == nums.length) return;

        // 枚举 nums[pos] 使用多少个
        for(int k = 0; k * nums[pos] + sum <= aim; k++)
        {
            if(k != 0) path.add(nums[pos]);
            dfs(nums, pos + 1, sum + k * nums[pos]);
        }

        // 恢复现场
        for(int k = 1; k * nums[pos] + sum <= aim; k++)
        {
            path.remove(path.size() - 1);
        }
```

```
38      }
39  }
```

Java 运行结果：

21. 字母大小写全排列（medium）

1. **题目链接：784. 字母大小写全排列**

2. **题目描述：**

> 给定一个字符串 s ，通过将字符串 s 中的每个字母转变大小写，我们可以获得一个新的字符串。
>
> 返回 所有可能得到的字符串集合 。以 任意顺序 返回输出。

- 示例 1：

> 输入：s = "a1b2"
>
> 输出：["a1b2", "a1B2", "A1b2", "A1B2"]

- 示例 2:

> 输入: s = "3z4"
>
> 输出: ["3z4","3Z4"]

- 提示:

> 1 <= s.length <= 12
>
> s 由小写英文字母、大写英文字母和数字组成

3. **解法：**

算法思路：

只需要对英文字母进行处理，处理每个元素时存在三种情况：

1. 不进行处理；
2. 若当前字母是英文字母并且是大写，将其修改为小写；
3. 若当前字母是英文字母并且是小写，将其修改为大写。

递归函数设计：void dfs(int step)

参数：step（当前需要处理的位置）；

返回值：无；

函数作用：查找所有可能的字符串集合，并将其记录在答案列表。

从前往后按序进行递归，**递归流程如下：**

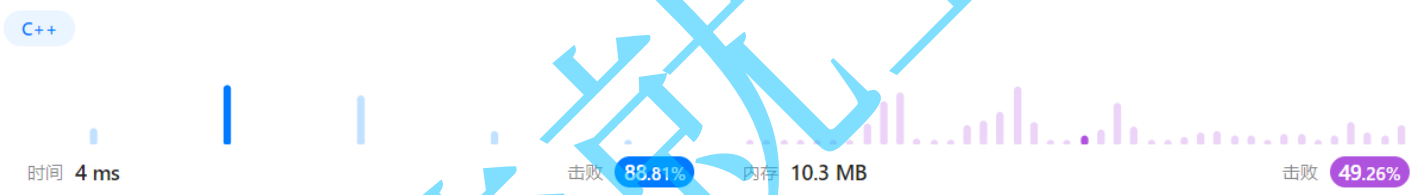
1. 递归结束条件：当前需要处理的元素下标越界，表示处理完毕，记录当前状态并返回；
2. 对当前元素不进行任何处理，直接递归下一位元素；
3. 判断当前元素是否为小写字母，若是，将其修改为大写字母并递归下一个元素，递归结束时撤销修改操作；
4. 判断当前元素是否为大写字母，若是，将其修改为小写字母并递归下一个元素，递归结束时撤销修改操作；

C++ 算法代码：

```cpp
class Solution
{
    string path;
    vector<string> ret;

public:
    vector<string> letterCasePermutation(string s)
    {
        dfs(s, 0);
        return ret;
    }

    void dfs(string& s, int pos)
    {
        if(pos == s.length())
        {
            ret.push_back(path);
            return;
        }

        char ch = s[pos];
        // 不改变
        path.push_back(ch);
        dfs(s, pos + 1);
        path.pop_back(); // 恢复现场

```

```
        // 改变
        if(ch < '0' || ch > '9')
        {
            char tmp = change(ch);
            path.push_back(tmp);
            dfs(s, pos + 1);
            path.pop_back(); // 恢复现场
        }
    }

    char change(char ch)
    {
        if(ch >= 'a' && ch <= 'z') ch -= 32;
        else ch += 32;
        return ch;
    }
};
```

C++ 运行结果：



Java 算法代码：

```
class Solution
{
    StringBuffer path;
    List<String> ret;

    public List<String> letterCasePermutation(String s)
    {
        path = new StringBuffer();
        ret = new ArrayList<>();

        dfs(s, 0);
        return ret;
    }

    public void dfs(String s, int pos)
    {
```

```
17          if(pos == s.length())
18          {
19              ret.add(path.toString());
20              return;
21          }
22
23          char ch = s.charAt(pos);
24          // 不改变
25          path.append(ch);
26          dfs(s, pos + 1);
27          path.deleteCharAt(path.length() - 1); // 恢复现场
28
29          // 改变
30          if(ch < '0' || ch > '9')
31          {
32              char tmp = change(ch);
33              path.append(tmp);
34              dfs(s, pos + 1);
35              path.deleteCharAt(path.length() - 1); // 恢复现场
36          }
37      }
38
39      public char change(char ch)
40      {
41          if(ch >= 'a' && ch <= 'z') return ch -= 32;
42          else return ch += 32;
43      }
44  }
```

Java 运行结果：

22. 优美的排列（medium）

1. **题目链接：526. 优美的排列**

2. **题目描述：**

> 假设有从 1 到 n 的 n 个整数。用这些整数构造一个数组 perm（下标从 1 开始），只要满足下述条件 之一 ，该数组就是一个 优美的排列 ：

> perm[i] 能够被 i 整除
>
> i 能够被 perm[i] 整除
>
> 给你一个整数 n ，返回可以构造的 优美排列 的 数量 。

- 示例 1：

> 输入：n = 2
>
> 输出：2
>
> 解释：
>
> 第 1 个优美的排列是 [1,2]：
>
> - perm[1] = 1 能被 i = 1 整除
> - perm[2] = 2 能被 i = 2 整除
>
> 第 2 个优美的排列是 [2,1]:
>
> - perm[1] = 2 能被 i = 1 整除
> - i = 2 能被 perm[2] = 1 整除

- 示例 2：

> 输入：n = 1
>
> 输出：1

- 提示：

> 1 <= n <= 15

3. **解法：**

算法思路：

我们需要在每一个位置上考虑所有的可能情况并且不能出现重复。通过深度优先搜索的方式，不断地枚举每个数在当前位置的可能性，并回溯到上一个状态，直到枚举完所有可能性，得到正确的结果。

我们需要定义一个变量 用来记录所有可能的排列数量，一个一维数组 visited 标记元素，然后从第一个位置开始进行递归；

递归函数设计：void backtrack(int index, int &n)

参数：index（当前需要处理的位置）；

返回值：无；

函数作用：在当前位置填入一个合理的数字，查找所有满足条件的排列。

递归流程如下：

1. 递归结束条件：当 index 等于 n 时，说明已经处理完了所有数字，将当前数组存入结果中；
2. 在每个递归状态中，枚举所有下标 x，若这个下标未被标记，并且满足题目条件之一：
   a. 将 visited[x] 标记为 1；
   b. 对第 index+1 个位置进行递归；
   c. 将 visited[x] 重新赋值为 0，表示回溯；

C++ 算法代码：

```cpp
class Solution
{
    bool check[16];
    int ret;

public:
    int countArrangement(int n)
    {
        dfs(1, n);
        return ret;
    }

    void dfs(int pos, int n)
    {
        if(pos == n + 1)
        {
            ret++;
            return;
        }

        for(int i = 1; i <= n; i++)
        {
            if(!check[i] && (pos % i == 0 || i % pos == 0))
            {
                check[i] = true;
                dfs(pos + 1, n);
                check[i] = false; // 恢复现场
            }
        }
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution
{
    boolean[] check;
    int ret;

    public int countArrangement(int n)
    {
        check = new boolean[n + 1];

        dfs(1, n);
        return ret;
    }

    public void dfs(int pos, int n)
    {
        if(pos == n + 1)
        {
            ret++;
            return;
        }

        for(int i = 1; i <= n; i++)
        {
            if(check[i] == false && (i % pos == 0 || pos % i == 0))
            {
                check[i] = true;
                dfs(pos + 1, n);
                check[i] = false; // 恢复现场
            }
        }
    }
}
```


Java 运行结果：

Java

时间 54 ms 击败 37.36%

内存 38.4 MB 击败 55.9%

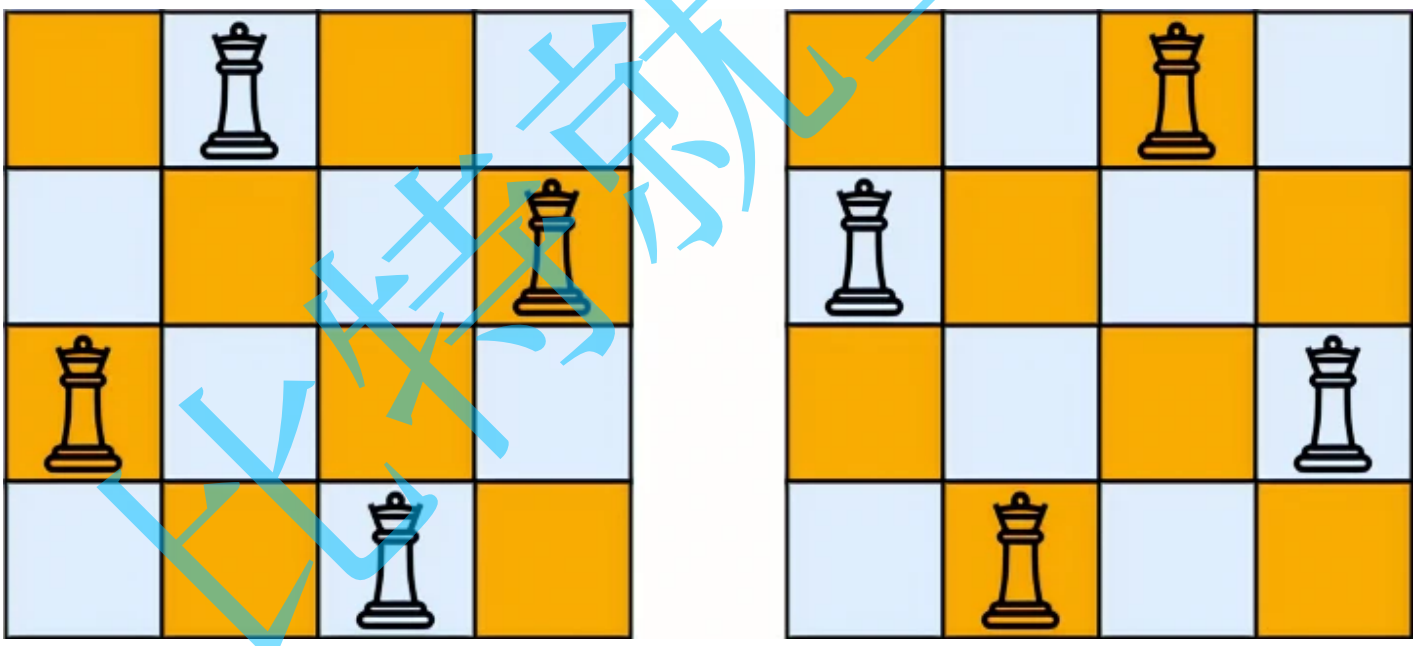
23. N 皇后（hard）

1. **题目链接：51. N 皇后**

2. **题目描述：**

> 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。
>
> n 皇后问题 研究的是如何将 n 个皇后放置在 n×n 的棋盘上，并且使皇后彼此之间不能相互攻击。
>
> 给你一个整数 n ，返回所有不同的 n 皇后问题 的解决方案。
>
> 每一种解法包含一个不同的 n 皇后问题 的棋子放置方案，该方案中 'Q' 和 '.' 分别代表了皇后和空位。

- 示例 1：



> 输入：n = 4
>
> 输出：[[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]
>
> 解释：如上图所示，4 皇后问题存在两个不同的解法。

- 示例 2：

> 输入：n = 1
>
> 输出：[["Q"]]

- 提示：

> 1 <= n <= 9

3. **解法：**

算法思路：

首先，我们在第一行放置第一个皇后，然后遍历棋盘的第二行，在可行的位置放置第二个皇后，然后再遍历第三行，在可行的位置放置第三个皇后，以此类推，直到放置了 n 个皇后为止。

我们需要用一个数组来记录每一行放置的皇后的列数。在每一行中，我们尝试放置一个皇后，并检查是否会和前面已经放置的皇后冲突。如果没有冲突，我们就继续递归地放置下一行的皇后，直到所有的皇后都放置完毕，然后把这个方案记录下来。

在检查皇后是否冲突时，我们可以用一个数组来记录每一列是否已经放置了皇后，并检查当前要放置的皇后是否会和已经放置的皇后冲突。对于对角线，我们可以用两个数组来记录从左上角到右下角的每一条对角线上是否已经放置了皇后，以及从右上角到左下角的每一条对角线上是否已经放置了皇后。

- 对于对角线是否冲突的判断可以通过以下流程解决：

1. 从左上到右下：相同对角线的行列之差相同；
2. 从右上到左下：相同对角线的行列之和相同。

因此，我们需要创建用于存储解决方案的二维字符串数组 `solutions`，用于存储每个皇后的位置的一维整数数组 `queens`，以及用于记录每一列和对角线上是否已经有皇后的布尔型数组 `columns`、`diagonals1` 和 `diagonals2`。

递归函数设计：void dfs(vector<vector<string>> &solutions, vector<int> &queens, int &n, int row, vector<bool> &columns, vector<bool> &diagonals1, vector<bool> &diagonals2)

参数：row（当前需要处理的行数）；

返回值：无；

函数作用：在当前行放入一个不发生冲突的皇后，查找所有可行的方案使得放置 n 个皇后后不发生冲突。

递归函数流程如下：

1. 结束条件：如果 `row` 等于 `n`，则表示已经找到一组解决方案，此时将每个皇后的位置存储到字符串数组 `board` 中，并将 `board` 存储到 `solutions` 数组中，然后返回；
2. 枚举当前行的每一列，判断该列、两个对角线上是否已经有皇后：
    a. 如果有皇后，则继续枚举下一列；
    b. 否则，在该位置放置皇后，并将 `columns`、`diagonals1` 和 `diagonals2` 对应的位置设为 `true`，表示该列和对角线上已经有皇后：

i. 递归调用 `dfs` 函数，搜索下一行的皇后位置。如果该方案递归结束，则在回溯时需要将 `columns` 、 `diagonals1` 和 `diagonals2` 对应的位置设为 `false` ，然后继续枚举下一列；

C++ 算法代码：

```
class Solution
{
    bool checkCol[10], checkDig1[20], checkDig2[20];
    vector<vector<string>> ret;
    vector<string> path;
    int n;

public:
    vector<vector<string>> solveNQueens(int _n)
    {
        n = _n;
        path.resize(n);
        for(int i = 0; i < n; i++)
            path[i].append(n, '.');

        dfs(0);
        return ret;
    }

    void dfs(int row)
    {
        if(row == n)
        {
            ret.push_back(path);
            return;
        }

        for(int col = 0; col < n; col++) // 尝试在这一行放皇后
        {
            // 剪枝
            if(!checkCol[col] && !checkDig1[row - col + n] && !checkDig2[row + col])
            {
                path[row][col] = 'Q';
                checkCol[col] = checkDig1[row - col + n] = checkDig2[row + col] = true;
                dfs(row + 1);
                // 恢复现场
                path[row][col] = '.';
```

```
                checkCol[col] = checkDig1[row - col + n] = checkDig2[row +
col] = false;
            }
        }
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution
{
    boolean[] checkCol, checkDig1, checkDig2;
    List<List<String>> ret;
    char[][] path;
    int n;

    public List<List<String>> solveNQueens(int _n)
    {
        n = _n;
        checkCol = new boolean[n];
        checkDig1 = new boolean[n * 2];
        checkDig2 = new boolean[n * 2];
        ret = new ArrayList<>();
        path = new char[n][n];

        for(int i = 0; i < n; i++)
            Arrays.fill(path[i], '.');

        dfs(0);
        return ret;
    }

    public void dfs(int row)
    {
        if(row == n)
        {
```

```
            // 添加结果
            List<String> tmp = new ArrayList<>();
            for(int i = 0; i < n; i++)
            {
                tmp.add(new String(path[i]));
            }
            ret.add(new ArrayList<>(tmp));
        }

        for(int col = 0; col < n; col++)
        {
            // 判断能不能放
            if(checkCol[col] == false && checkDig1[row - col + n] == false &&
checkDig2[row + col] == false)
            {
                path[row][col] = 'Q';
                checkCol[col] = checkDig1[row - col + n] = checkDig2[row +
col] = true;
                dfs(row + 1);
                // 恢复现场
                path[row][col] = '.';
                checkCol[col] = checkDig1[row - col + n] = checkDig2[row +
col] = false;
            }
        }
    }
}
```

Java 运行结果：

24. 有效的数独（medium）

1. **题目链接：36. 有效的数独**

2. **题目描述：**

> 请你判断一个 9 x 9 的数独是否有效。只需要 根据以下规则 ，验证已经填入的数字是否有效即可。
>
> 数字 1-9 在每一行只能出现一次。
>
> 数字 1-9 在每一列只能出现一次。

数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。（请参考示例图）

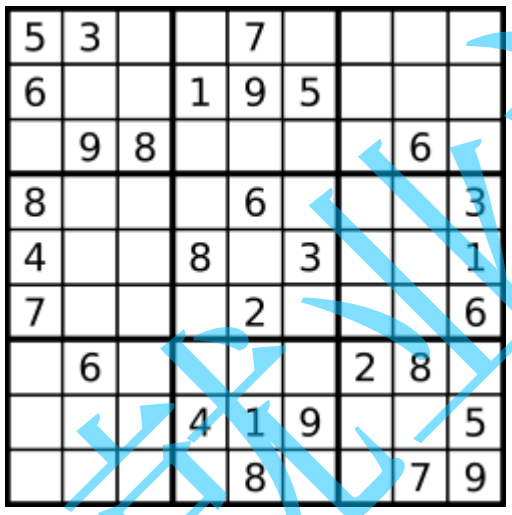
注意：

一个有效的数独（部分已被填充）不一定是可解的。

只需要根据以上规则，验证已经填入的数字是否有效即可。

空白格用 '.' 表示。

示例 1：



输入：

board =

[["5","3",".",".","7",".",".",".","."]

,["6",".",".","1","9","5",".",".","."]

,[".","9","8",".",".",".",".","6","."]

,["8",".",".",".","6",".",".",".","3"]

,["4",".",".","8",".","3",".",".","1"]

,["7",".",".",".","2",".",".",".","6"]

,[".","6",".",".",".",".","2","8","."]

,[".",".",".","4","1","9",".",".","5"]

,[".",".",".",".","8",".",".","7","9"]]

输出：true

示例 2：

输入：

board =

[["8","3",".",".","7",".",".",".","."]

,["6",".",".","1","9","5",".",".","."]

,[".","9","8",".",".",".",".","6","."]

,["8",".",".",".","6",".",".",".","3"]

,["4",".",".","8",".","3",".",".","1"]

,["7",".",".",".","2",".",".",".","6"]

,[".","6",".",".",".",".","2","8","."]

,[".",".",".","4","1","9",".",".","5"]

,[".",".",".",".","8",".",".","7","9"]]

输出：false

解释：除了第一行的第一个数字从 5 改为 8 以外，空格内其他数字均与 示例1 相同。 但由于位于左上角的 3x3 宫内有两个 8 存在, 因此这个数独是无效的。

提示：

board.length == 9

board[i].length == 9

board[i][j] 是一位数字（1-9）或者 '.'

3. **解法：**

算法思路：

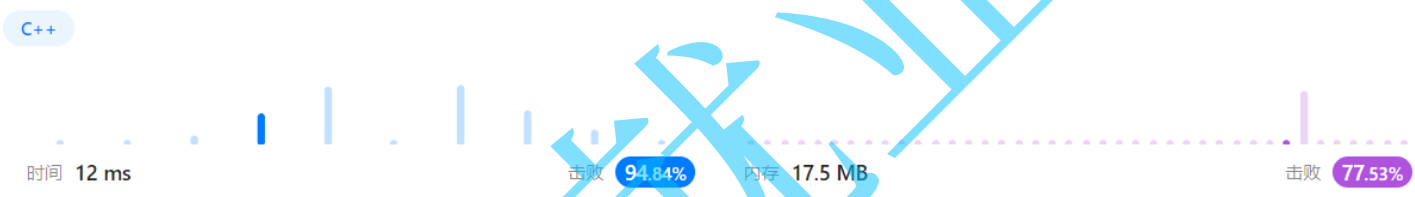
创建三个数组标记行、列以及 `3*3` 小方格中是否出现 `1~9` 之间的数字即可。

C++ 算法代码：

```
class Solution
{
    bool row[9][10];
    bool col[9][10];
    bool grid[3][3][10];

public:
    bool isValidSudoku(vector<vector<char>>& board)
    {
```

```
        for(int i = 0; i < 9; i++)
            for(int j = 0; j < 9; j++)
            {
                if(board[i][j] != '.')
                {
                    int num = board[i][j] - '0';
                    // 是否是有效的
                    if(row[i][num] || col[j][num] || grid[i / 3][j / 3][num])
                        return false;
                    row[i][num] = col[j][num] = grid[i / 3][j / 3][num] = true;
                }
            }
        return true;
    }
};
```

C++ 代码结果：



Java 算法代码：

```
class Solution
{
    boolean[][] row, col;
    boolean[][][] grid;

    public boolean isValidSudoku(char[][] board)
    {
        row = new boolean[9][10];
        col = new boolean[9][10];
        grid = new boolean[3][3][10];

        for(int i = 0; i < 9; i++)
            for(int j = 0; j < 9; j++)
            {
                if(board[i][j] != '.')
                {
                    int num = board[i][j] - '0';
                    // 是否有效
```

```
19                     if(row[i][num] || col[j][num] || grid[i / 3][j / 3][num])
20                         return false;
21                     row[i][num] = col[j][num] = grid[i / 3][j / 3][num] = true;
22                 }
23             }
24         return true;
25     }
26 }
```

Java 运行结果：

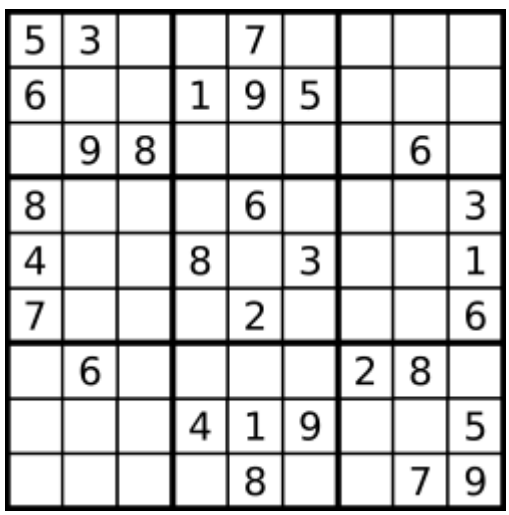
25. 解数独（hard）

1. **题目链接：37. 解数独**

2. **题目描述：**

> 编写一个程序，通过填充空格来解决数独问题。
>
> 数独的解法需 遵循如下规则：
>
> 数字 1-9 在每一行只能出现一次。
>
> 数字 1-9 在每一列只能出现一次。
>
> 数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。（请参考示例图）
>
> 数独部分空格内已填入了数字，空白格用 '.' 表示。

- 示例 1：

> 
>

> 输入：board = [["5",".","3",".",".","7",".",".",".","."],["6",".",".","1","9","5",".",".","."],[".","9","8",".",".",".",".",".","6","."],["8",".",".",".","6",".",".",".",".","3"],["4",".",".","8",".","3",".",".","1"],["7",".",".",".",".","2",".",".",".","6"],[".","6",".",".",".",".",".","2","8","."],[".",".",".","4","1","9",".",".","5"],[".",".",".",".","8",".",".","7","9"]]
>
> 输出：[["5","3","4","6","7","8","9","1","2"],["6","7","2","1","9","5","3","4","8"],["1","9","8","3","4","2","5","6","7"],["8","5","9","7","6","1","4","2","3"],["4","2","6","8","5","3","7","9","1"],["7","1","3","9","2","4","8","5","6"],["9","6","1","5","3","7","2","8","4"],["2","8","7","4","1","9","6","3","5"],["3","4","5","2","8","6","1","7","9"]]
>
> 解释：输入的数独如上图所示，唯一有效的解决方案如下所示：

- 提示：

> board.length == 9
>
> board[i].length == 9
>
> board[i][j] 是一位数字或者 '.'
>
> 题目数据 保证 输入数独仅有一个解

3. **解法：**

算法思路：

为了存储每个位置的元素，我们需要定义一个二维数组。首先，我们记录所有已知的数据，然后遍历所有需要处理的位置，并遍历数字 1~9。对于每个位置，我们检查该数字是否可以存放在该位置，同时检查行、列和九宫格是否唯一。

我们可以使用一个二维数组来记录每个数字在每一行中是否出现，一个二维数组来记录每个数字在每一列中是否出现。对于九宫格，我们可以以**行和列除以 3 得到的商作为九宫格的坐标**，并使用一个三维数组来记录每个数字在每一个九宫格中是否出现。在检查是否存在冲突时，只需检查行、列和九宫格里对应的数字是否已被标记。如果数字至少有一个位置（行、列、九宫格）被标记，则存在冲突，因此不能在该位置放置当前数字。

- 特别地，在本题中，我们需要直接修改给出的数组，因此在找到一种可行的方法时，应该停止递归，以防止正确的方法被覆盖。

初始化定义：

1. 定义行、列、九宫格标记数组以及找到可行方法的标记变量，将它们初始化为 false。
2. 定义一个数组来存储每个需要处理的位置。
3. 将题目给出的所有元素的行、列以及九宫格坐标标记为 true。
4. 将所有需要处理的位置存入数组。

递归函数设计：void dfs(vector<vector<char>>& board, int pos)

参数：pos（当前需要处理的坐标）；

返回值：无；

函数作用：在当前坐标填入合适数字，查找数独答案。

递归流程如下：

1. 结束条件：已经处理完所有需要处理的元素。如果找到了可行的解决方案，则将标记变量更新为 true 并返回。
2. 获取当前需要处理的元素的行列值。
3. 遍历数字 1~9。如果当前数字可以填入当前位置，并且标记变量未被赋值为 true，则将当前位置的行、列以及九宫格坐标标记为 true，将当前数字赋值给 board 数组中的相应位置元素，然后对下一个位置进行递归。
4. 递归结束时，撤回标记。

C++ 算法代码：

```
class Solution
{
    bool row[9][10], col[9][10], grid[3][3][10];

public:
    void solveSudoku(vector<vector<char>>& board)
    {
        // 初始化
        for(int i = 0; i < 9; i++)
        {
            for(int j = 0; j < 9; j++)
            {
```

```
13                  if(board[i][j] != '.')
14                  {
15                      int num = board[i][j] - '0';
16                      row[i][num] = col[j][num] = grid[i / 3][j / 3][num] = true;
17                  }
18              }
19          }
20
21          dfs(board);
22      }
23
24      bool dfs(vector<vector<char>>& board)
25      {
26          for(int i = 0; i < 9; i++)
27          {
28              for(int j = 0; j < 9; j++)
29              {
30                  if(board[i][j] == '.')
31                  {
32                      // 填数
33                      for(int num = 1; num <= 9; num++)
34                      {
35                          if(!row[i][num] && !col[j][num] && !grid[i / 3][j / 3]
    [num])
36                          {
37                              board[i][j] = '0' + num;
38                              row[i][num] = col[j][num] = grid[i / 3][j / 3]
    [num] = true;
39                              if(dfs(board) == true) return true; // 重点理解
40                              // 恢复现场
41                              board[i][j] = '.';
42                              row[i][num] = col[j][num] = grid[i / 3][j / 3]
    [num] = false;
43                          }
44                      }
45                      return false; // 重点理解
46                  }
47              }
48          }
49          return true; // 重点理解
50      }
51  };
```

C++ 运行结果：

C++

时间 8 ms 击败 80.17%

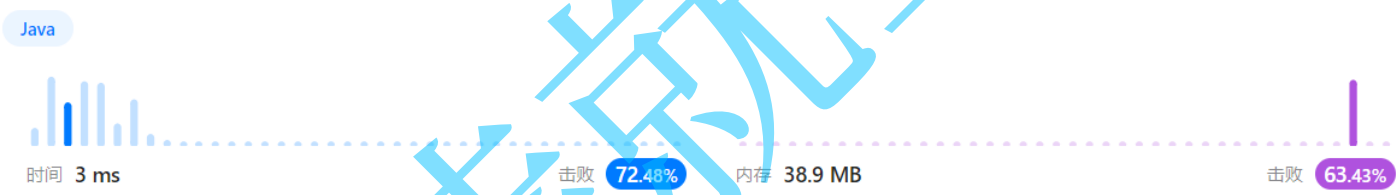
内存 6.2 MB 击败 57.91%

Java 算法代码：

```java
class Solution
{
    boolean[][] row, col;
    boolean[][][] grid;

    public void solveSudoku(char[][] board)
    {
        row = new boolean[9][10];
        col = new boolean[9][10];
        grid = new boolean[3][3][10];

        // 初始化
        for(int i = 0; i < 9; i++)
            for(int j = 0; j < 9; j++)
                if(board[i][j] != '.')
                {
                    int num = board[i][j] - '0';
                    row[i][num] = col[j][num] = grid[i / 3][j / 3][num] = true;
                }

        dfs(board);
    }

    public boolean dfs(char[][] board)
    {
        for(int i = 0; i < 9; i++)
        {
            for(int j = 0; j < 9; j++)
            {
                if(board[i][j] == '.')
                {
                    // 填数
                    for(int num = 1; num <= 9; num++)
                    {
                        // 剪枝
                        if(!row[i][num] && !col[j][num] && !grid[i / 3][j / 3]
[num])
```

```
37                          {
38                              board[i][j] = (char)('0' + num);
39                              row[i][num] = col[j][num] = grid[i / 3][j / 3]
   [num] = true;
40                              if(dfs(board) == true) return true; // 重点理解
41                              // 恢复现场
42                              board[i][j] = '.';
43                              row[i][num] = col[j][num] = grid[i / 3][j / 3]
   [num] = false;
44                          }
45                      }
46                      return false; // 重点理解
47                  }
48              }
49          }
50          return true; // 重点理解
51      }
52  }
```

Java 运行结果：

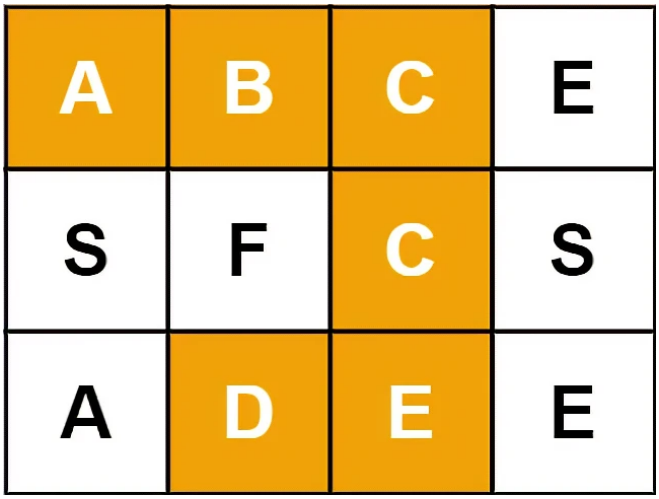


26. 单词搜索（medium）

1. **题目链接：79. 单词搜索**
2. **题目描述：**

> 给定一个 m x n 二维字符网格 board 和一个字符串单词 word 。如果 word 存在于网格中，返回 true ；否则，返回 false 。
>
> 单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

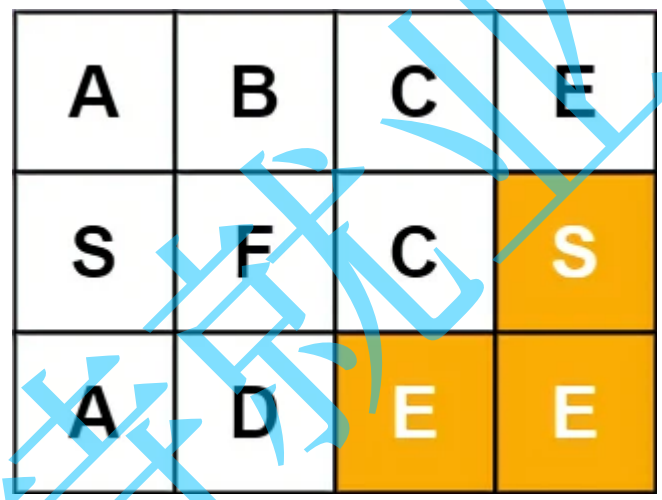
- 示例 1：



输入：board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"

输出：true

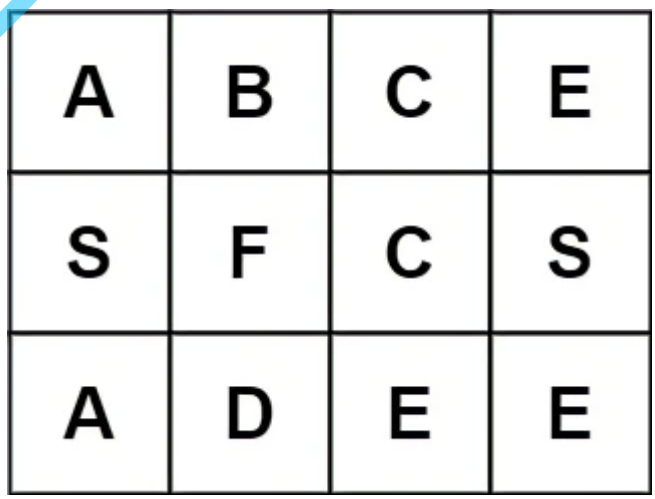
- 示例 2：



输入：board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "SEE"

输出：true

- 示例 3：



输入：board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCB"

> 输出：false

- 提示：

> m == board.length
>
> n = board[i].length
>
> 1 <= m, n <= 6
>
> 1 <= word.length <= 15
>
> board 和 word 仅由大小写英文字母组成

3. **解法：**

算法思路：

我们需要假设每个位置的元素作为第一个字母，然后向相邻的四个方向进行递归，并且不能出现重复使用同一个位置的元素。通过深度优先搜索的方式，不断地枚举相邻元素作为下一个字母出现的可能性，并在递归结束时回溯，直到枚举完所有可能性，得到正确的结果。

递归函数设计：bool dfs(int x, int y, int step, vector<vector<char>>& board, string word, vector<vector<bool>>& vis, int &n, int &m, int &len)

参数：x（当前需要进行处理的元素横坐标），y（当前需要进行处理的元素横坐标），step（当前已经处理的元素个数），word（当前的字符串状态）；

返回值：当前坐标元素作为字符串中下标 step 的元素出现是否可以找到成立的字符串。

函数作用：判断当前坐标的元素作为字符串中下标 step 的元素出现时，向四个方向传递，查找是否存在路径结果与字符串相同。

递归函数流程：

1. 遍历每个位置，标记当前位置并将当前位置的字母作为首字母进行递归，并且在回溯时撤回标记。
2. 在每个递归的状态中，我们维护一个步数 step，表示当前已经处理了几个字母。
    - 若当前位置的字母与字符串中的第 step 个字母不相等，则返回 false。
    - 若当前 step 的值与字符串长度相等，表示存在一种路径使得 word 成立，返回 true。
3. 对当前位置的上下左右四个相邻位置进行递归，若递归结果为 true，则返回 true。
4. 若相邻的四个位置的递归结果都为 false，则返回 false。

- 特别地，如果使用将当前遍历到的字符赋值为空格，并在回溯时恢复为原来的字母的方法，则在递归时不会重复遍历当前元素，可达到不使用标记数组的目的。

C++ 算法代码：

```cpp
class Solution
{
    bool vis[7][7];
    int m, n;

public:
    bool exist(vector<vector<char>>& board, string word)
    {
        m = board.size(), n = board[0].size();
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
            {
                if(board[i][j] == word[0])
                {
                    vis[i][j] = true;
                    if(dfs(board, i, j, word, 1)) return true;
                    vis[i][j] = false;
                }
            }
        return false;
    }

    int dx[4] = {0, 0, -1, 1};
    int dy[4] = {1, -1, 0, 0};

    bool dfs(vector<vector<char>>& board, int i, int j, string& word, int pos)
    {
        if(pos == word.size()) return true;

        // 向量的方式，定义上下左右四个位置
        for(int k = 0; k < 4; k++)
        {
            int x = i + dx[k], y = j + dy[k];
            if(x >= 0 && x < m && y >= 0 && y < n && !vis[x][y] && board[x][y] == word[pos])
            {
                vis[x][y] = true;
                if(dfs(board, x, y, word, pos + 1)) return true;
                vis[x][y] = false;
            }
        }
        return false;
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution
{
    boolean[][] vis;
    int m, n;
    char[] word;

    public boolean exist(char[][] board, String _word)
    {
        m = board.length; n = board[0].length;
        word = _word.toCharArray();
        vis = new boolean[m][n];

        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
            {
                if(board[i][j] == word[0])
                {
                    vis[i][j] = true;
                    if(dfs(board, i, j, 1)) return true;
                    vis[i][j] = false;
                }
            }
        return false;
    }

    int[] dx = {0, 0, 1, -1};
    int[] dy = {1, -1, 0, 0};

    public boolean dfs(char[][] board, int i, int j, int pos)
    {
        if(pos == word.length)
        {
            return true;
        }

```

```
36          // 上下左右去匹配 word[pos]
37          // 利用向量数组，一个 for 搞定上下左右四个方向
38          for(int k = 0; k < 4; k++)
39          {
40              int x = i + dx[k], y = j + dy[k];
41              if(x >= 0 && x < m && y >= 0 && y < n && !vis[x][y] && board[x][y]
    == word[pos])
42              {
43                  vis[x][y] = true;
44                  if(dfs(board, x, y, pos + 1)) return true;
45                  vis[x][y] = false;
46              }
47          }
48          return false;
49      }
50  }
```

Java 运行结果：

27. 黄金矿工（medium）

1. **题目链接：1219. 黄金矿工**
2. **题目描述：**

> 你要开发一座金矿，地质勘测学家已经探明了这座金矿中的资源分布，并用大小为 m * n 的网格 grid 进行了标注。每个单元格中的整数就表示这一单元格中的黄金数量；如果该单元格是空的，那么就是 0。
>
> 为了使收益最大化，矿工需要按以下规则来开采黄金：
>
> - 每当矿工进入一个单元，就会收集该单元格中的所有黄金。
> - 矿工每次可以从当前位置向上下左右四个方向走。
> - 每个单元格只能被开采（进入）一次。
> - 不得开采（进入）黄金数目为 0 的单元格。
> - 矿工可以从网格中 任意一个 有黄金的单元格出发或者是停止。

示例 1：

输入：grid = [[0,6,0],[5,8,7],[0,9,0]]

输出：24

解释：

[[0,6,0],

[5,8,7],

[0,9,0]]

一种收集最多黄金的路线是：9 -> 8 -> 7。

示例 2：

输入：grid = [[1,0,7],[2,0,6],[3,4,5],[0,3,0],[9,0,20]]

输出：28

解释：

[[1,0,7],

[2,0,6],

[3,4,5],

[0,3,0],

[9,0,20]]

一种收集最多黄金的路线是：1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7。

提示：

1 <= grid.length, grid[i].length <= 15

0 <= grid[i][j] <= 100

最多 25 个单元格中有黄金。

3. **解法：**

算法思路：

枚举矩阵中所有的位置当成起点，来一次深度优先遍历，统计出所有情况下能收集到的黄金数的最大值即可。

C++ 算法代码：

```cpp
class Solution
{
    bool vis[16][16];
    int dx[4] = {0, 0, 1, -1};
    int dy[4] = {1, -1, 0, 0};
    int m, n;
    int ret;

public:
    int getMaximumGold(vector<vector<int>>& g)
    {
        m = g.size(), n = g[0].size();
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
            {
                if(g[i][j])
                {
                    vis[i][j] = true;
                    dfs(g, i, j, g[i][j]);
                    vis[i][j] = false;
                }
            }
        return ret;
    }

    void dfs(vector<vector<int>>& g, int i, int j, int path)
    {
        ret = max(ret, path);

        for(int k = 0; k < 4; k++)
        {
            int x = i + dx[k], y = j + dy[k];
            if(x >= 0 && x < m && y >= 0 && y < n && !vis[x][y] && g[x][y])
            {
                vis[x][y] = true;
                dfs(g, x, y, path + g[x][y]);
                vis[x][y] = false;
            }
        }
    }
};
```


C++ 代码结果：

C++

时间 156 ms 击败 84.79%

内存 7.1 MB 击败 99.15%

Java 算法代码：

```java
class Solution
{
    boolean[][] vis;
    int m, n;
    int[] dx = {0, 0, -1, 1};
    int[] dy = {1, -1, 0, 0};
    int ret;

    public int getMaximumGold(int[][] g)
    {
        m = g.length; n = g[0].length;
        vis = new boolean[m][n];

        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
            {
                if(g[i][j] != 0)
                {
                    vis[i][j] = true;
                    dfs(g, i, j, g[i][j]);
                    vis[i][j] = false;
                }
            }
        return ret;
    }

    public void dfs(int[][] g, int i, int j, int path)
    {
        ret = Math.max(ret, path);

        for(int k = 0; k < 4; k++)
        {
            int x = i + dx[k], y = j + dy[k];
            if(x >= 0 && x < m && y >= 0 && y < n && !vis[x][y] && g[x][y] != 0)
            {
                vis[x][y] = true;
```

```
37                 dfs(g, x, y, path + g[x][y]);
38                 vis[x][y] = false;
39             }
40         }
41     }
42 }
```

Java 运行结果：

28. 不同路径 III（hard）

1. **题目链接：980. 不同路径 III**

2. **题目描述：**

> 在二维网格 grid 上，有 4 种类型的方格：
>
> 1 表示起始方格。且只有一个起始方格。
>
> 2 表示结束方格，且只有一个结束方格。
>
> 0 表示我们可以走过的空方格。
>
> -1 表示我们无法跨越的障碍。
>
> 返回在四个方向（上、下、左、右）上行走时，从起始方格到结束方格的不同路径的数目。
>
> 每一个无障碍方格都要通过一次，但是一条路径中不能重复通过同一个方格。

- 示例 1：

> 输入：[[1,0,0,0],[0,0,0,0],[0,0,2,-1]]
>
> 输出：2
>
> 解释：我们有以下两条路径：
>
> 1. (0,0),(0,1),(0,2),(0,3),(1,3),(1,2),(1,1),(1,0),(2,0),(2,1),(2,2)
> 2. (0,0),(1,0),(2,0),(2,1),(1,1),(0,1),(0,2),(0,3),(1,3),(1,2),(2,2)

- 示例 2：

> 输入：[[1,0,0,0],[0,0,0,0],[0,0,0,2]]
>
> 输出：4

> 解释：我们有以下四条路径：
>
> 1. (0,0),(0,1),(0,2),(0,3),(1,3),(1,2),(1,1),(1,0),(2,0),(2,1),(2,2),(2,3)
> 2. (0,0),(0,1),(1,1),(1,0),(2,0),(2,1),(2,2),(1,2),(0,2),(0,3),(1,3),(2,3)
> 3. (0,0),(1,0),(2,0),(2,1),(2,2),(1,2),(1,1),(0,1),(0,2),(0,3),(1,3),(2,3)
> 4. (0,0),(1,0),(2,0),(2,1),(1,1),(0,1),(0,2),(0,3),(1,3),(1,2),(2,2),(2,3)

- 示例 3：

> 输入：[[0,1],[2,0]]
>
> 输出：0
>
> 解释：
>
> 没有一条路能完全穿过每一个空的方格一次。
>
> 请注意，起始和结束方格可以位于网格中的任意位置。

- 提示：

> 1 <= grid.length * grid[0].length <= 20

3. **解法：**

算法思路：

对于四个方向，我们可以定义一个二维数组 `next`，大小为 `4`，每一维存储四个方向的坐标偏移量（详见代码）。题目要求到达目标位置时所有无障碍方格都存在路径中，我们可以定义一个变量记录 `num` 当前状态中剩余的未走过的无障碍方格个数，则当我们走到目标地点时只需要判断 `num` 是否为 `0` 即可。在移动时需要判断是否越界。

递归函数设计：void dfs(vector<vector<int>>& grid, int x, int y, int num)

参数：x，y（当前需要处理元素的坐标），num（当前剩余无障碍方格个数）；

返回值：无；

函数作用：判断当前位置的四个方向是否可以添加至当前状态，查找在满足条件下从起始方格到结束方格的不同路径的数目。

递归流程如下：

1. 递归结束条件：当前位置的元素值为 2，若此时可走的位置数量 num 的值为 0，则 cnt 的值加一；
2. 遍历四个方向，若移动后未越界，无障碍并且未被标记，则标记当前位置，并递归移动后的位置，在回溯时撤销标记操作。

C++ 算法代码：

```cpp
class Solution
{
    bool vis[21][21];
    int dx[4] = {1, -1, 0, 0};
    int dy[4] = {0, 0, 1, -1};
    int ret;
    int m, n, step;

public:
    int uniquePathsIII(vector<vector<int>>& grid)
    {
        m = grid.size(), n = grid[0].size();

        int bx = 0, by = 0;
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
                if(grid[i][j] == 0) step++;
                else if(grid[i][j] == 1)
                {
                    bx = i;
                    by = j;
                }

        step += 2;
        vis[bx][by] = true;
        dfs(grid, bx, by, 1);
        return ret;
    }

    void dfs(vector<vector<int>>& grid, int i, int j, int count)
    {
        if(grid[i][j] == 2)
        {
            if(count == step) // 判断是否合法
                ret++;
            return;
        }

        for(int k = 0; k < 4; k++)
        {
            int x = i + dx[k], y = j + dy[k];
            if(x >= 0 && x < m && y >= 0 && y < n && !vis[x][y] && grid[x][y]
!= -1)
            {
                vis[x][y] = true;
                dfs(grid, x, y, count + 1);
```

```
                vis[x][y] = false;
            }
        }
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution
{
    boolean[][] vis;
    int m, n, step;
    int ret;
    int[] dx = {0, 0, 1, -1};
    int[] dy = {1, -1, 0, 0};

    public int uniquePathsIII(int[][] grid)
    {
        m = grid.length; n = grid[0].length;
        vis = new boolean[m][n];

        int bx = 0, by = 0;
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
                if(grid[i][j] == 0) step++;
                else if(grid[i][j] == 1)
                {
                    bx = i;
                    by = j;
                }
        step += 2;
        vis[bx][by] = true;
        dfs(grid, bx, by, 1);
        return ret;
    }

```

```java
    public void dfs(int[][] grid, int i, int j, int count)
    {
        if(grid[i][j] == 2)
        {
            if(count == step)
            {
                ret++;
            }
            return;
        }

        for(int k = 0; k < 4; k++)
        {
            int x = i + dx[k], y = j + dy[k];
            if(x >= 0 && x < m && y >= 0 && y < n && !vis[x][y] && grid[x][y] != -1)
            {
                vis[x][y] = true;
                dfs(grid, x, y, count + 1);
                vis[x][y] = false;
            }
        }

    }
}
```

Java 运行结果：

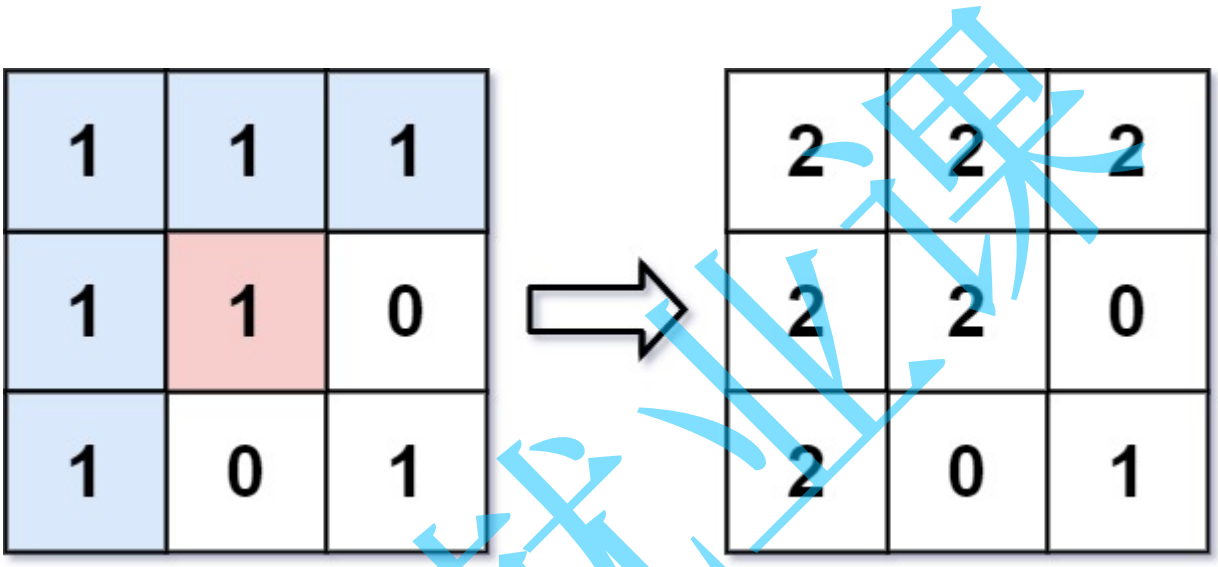
FloodFill 算法

29. 图像渲染（medium）

1. **题目链接：733. 图像渲染**

2. **题目描述：**

> 有一幅以 m x n 的二维整数数组表示的图画 image ，其中 image[i][j] 表示该图画的像素值大小。

> 你也被给予三个整数 sr，sc 和 newColor 。你应该从像素 image[sr][sc] 开始对图像进行 上色填充 。
>
> 为了完成 上色工作 ，从初始像素开始，记录初始坐标的 上下左右四个方向上 像素值与初始坐标相同的相连像素点，接着再记录这四个方向上符合条件的像素点与他们对应 四个方向上 像素值与初始坐标相同的相连像素点，……，重复该过程。将所有有记录的像素点的颜色值改为 newColor 。
>
> 最后返回 经过上色渲染后的图像 。
>
> 示例 1:
>
> 
>
>
> 输入: image = [[1,1,1],[1,1,0],[1,0,1]]，sr = 1, sc = 1, newColor = 2
>
> 输出: [[2,2,2],[2,2,0],[2,0,1]]
>
> 解析: 在图像的正中间，(坐标(sr,sc)=(1,1)),在路径上所有符合条件的像素点的颜色都被更改成 2。
>
> 注意，右下角的像素没有更改为 2，因为它不是在上下左右四个方向上与初始点相连的像素点。
>
> 示例 2:
>
> 输入: image = [[0,0,0],[0,0,0]], sr = 0, sc = 0, newColor = 2
>
> 输出: [[2,2,2],[2,2,2]]

3. **算法思路：**

可以利用「深搜」或者「宽搜」，遍历到与该点相连的所有「像素相同的点」，然后将其修改成指定的像素即可。

递归函数设计：

- 参数：

a. 原始矩阵；

b. 当前所在的位置；

c. 需要修改成的颜色。

- 函数体：

a. 先将该位置的颜色改成指定颜色（因为我们的判断，保证每次进入递归的位置都是需要修改的位置）；

b. 遍历四个方向上的位置：

- 如果当前位置合法，并且与初试颜色相同，就递归进去。

C++ 算法代码：

```
class Solution
{
    int dx[4] = {0, 0, 1, -1};
    int dy[4] = {1, -1, 0, 0};
    int m, n;
    int prev;

public:
    vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc,
int color)
    {
        if(image[sr][sc] == color) return image;

        m = image.size(), n = image[0].size();
        prev = image[sr][sc];
        dfs(image, sr, sc, color);
        return image;
    }

    void dfs(vector<vector<int>>& image, int i, int j, int color)
    {
        image[i][j] = color;

        for(int k = 0; k < 4; k++)
        {
            int x = i + dx[k], y = j + dy[k];
            if(x >= 0 && x < m && y >= 0 && y < n && image[x][y] == prev)
            {
                dfs(image, x, y, color);
            }
        }
```

```
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution
{
    int[] dx = {0, 0, 1, -1};
    int[] dy = {1, -1, 0, 0};
    int m, n;
    int prev;

    public int[][] floodFill(int[][] image, int sr, int sc, int color)
    {
        if(image[sr][sc] == color) return image;

        m = image.length; n = image[0].length;
        prev = image[sr][sc];
        dfs(image, sr, sc, color);
        return image;
    }

    public void dfs(int[][] image, int i, int j, int color)
    {
        image[i][j] = color;

        for(int k = 0; k < 4; k++)
        {
            int x = i + dx[k], y = j + dy[k];
            if(x >= 0 && x < m && y >= 0 && y < n && image[x][y] == prev)
            {
                dfs(image, x, y, color);
            }
        }
    }
}
```

Java 运行结果：

30. 岛屿数量（medium）

1. **题目链接：200. 岛屿数量**

2. **题目描述：**

> 给你一个由 '1'（陆地）和 '0'（水）组成的的二维网格，请你计算网格中岛屿的数量。
>
> 岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。
>
> 此外，你可以假设该网格的四条边均被水包围。
>
> 示例 1：
>
> 输入：grid = [
>
> ["1","1","1","1","0"],
>
> ["1","1","0","1","0"],
>
> ["1","1","0","0","0"],
>
> ["0","0","0","0","0"]
>
>]
>
> 输出：1
>
> 示例 2：
>
> 输入：grid = [
>
> ["1","1","0","0","0"],
>
> ["1","1","0","0","0"],
>
> ["0","0","1","0","0"],
>
> ["0","0","0","1","1"]
>
>]
>
> 输出：3

> 提示：
>
> `m == grid.length`
>
> `n == grid[i].length`
>
> `1 <= m, n <= 300`
>
> `grid[i][j]` 的值为 '0' 或 '1'

算法思路：

遍历整个矩阵，每次找到「一块陆地」的时候：

- 说明找到「一个岛屿」，记录到最终结果 `ret` 里面；
- 并且将这个陆地相连的所有陆地，也就是这块「岛屿」，全部「变成海洋」。这样的话，我们下次遍历到这块岛屿的时候，它「已经是海洋」了，不会影响最终结果。
- 其中「变成海洋」的操作，可以利用「深搜」和「宽搜」解决，其实就是 733. 图像渲染 这道题~

这样，当我们，遍历完全部的矩阵的时候，`ret` 存的就是最终结果。

解法（dfs）：

算法流程：

1. 初始化 `ret = 0`，记录目前找到的岛屿数量；
2. 双重循环遍历二维网格，每当遇到一块陆地，标记这是一个新的岛屿，然后将这块陆地相连的陆地全部变成海洋。

递归函数的设计：

1. 把当前格子标记为水；
2. 向上、下、左、右四格递归寻找陆地，只有在下标位置合理的情况下，才会进入递归：
   a. 下一个位置的坐标合理；
   b. 并且下一个位置是陆地

C++ 算法代码：

```
1 class Solution
2 {
```

```
    vector<vector<bool>> vis;
    int m, n;

public:
    int numIslands(vector<vector<char>>& grid)
    {
        m = grid.size(), n = grid[0].size();
        vis = vector<vector<bool>>(m, vector<bool>(n));

        int ret = 0;
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
            {
                if(!vis[i][j] && grid[i][j] == '1')
                {
                    ret++;
                    dfs(grid, i, j);
                }
            }
        return ret;
    }

    int dx[4] = {0, 0, -1, 1};
    int dy[4] = {1, -1, 0, 0};

    void dfs(vector<vector<char>>& grid, int i, int j)
    {
        vis[i][j] = true;
        for(int k = 0; k < 4; k++)
        {
            int x = i + dx[k], y = j + dy[k];
            if(x >= 0 && x < m && y >= 0 && y < n && !vis[x][y] && grid[x][y]
== '1')
            {
                dfs(grid, x, y);
            }
        }
    }
};
```

C++ 运行结果：

Java 算法代码：

```java
class Solution
{
    boolean[][] vis;
    int m, n;
    int[] dx = {0, 0, -1, 1};
    int[] dy = {1, -1, 0, 0};

    public int numIslands(char[][] grid)
    {
        m = grid.length; n = grid[0].length;
        vis = new boolean[m][n];

        int ret = 0;
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
            {
                if(!vis[i][j] && grid[i][j] == '1')
                {
                    ret++;
                    dfs(grid, i, j);
                }
            }
        return ret;
    }

    public void dfs(char[][] grid, int i, int j)
    {
        vis[i][j] = true;
        for(int k = 0; k < 4; k++)
        {
            int x = i + dx[k], y = j + dy[k];
            if(x >= 0 && x < m && y >= 0 && y < n && !vis[x][y] && grid[x][y]
== '1')
            {
                dfs(grid, x, y);
            }
        }
```

```
37         }
38     }
```

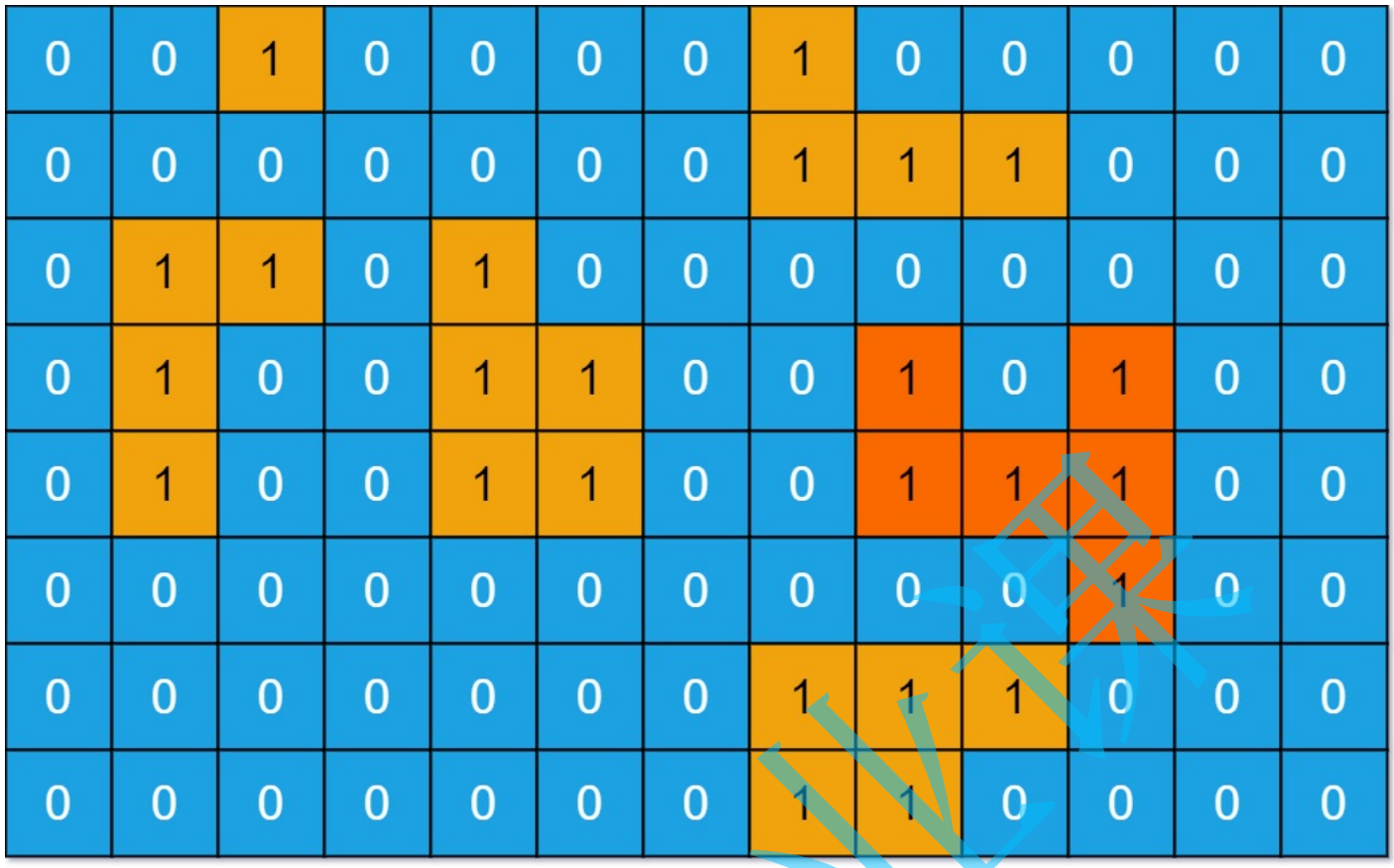
Java 运行结果：

31. 岛屿的最大面积（medium）

1. **题目链接：695. 岛屿的最大面积**

2. **题目描述：**

> 给你一个大小为 m x n 的二进制矩阵 grid 。
>
> 岛屿 是由一些相邻的 1 (代表土地) 构成的组合，这里的「相邻」要求两个 1 必须在 水平或者竖直的四个方向上 相邻。你可以假设 grid 的四个边缘都被 0（代表水）包围着。
>
> 岛屿的面积是岛上值为 1 的单元格的数目。
>
> 计算并返回 grid 中最大的岛屿面积。如果没有岛屿，则返回面积为 0 。
>
> 示例 1：



输入：

grid = [[0,0,1,0,0,0,0,1,0,0,0,0,0],[0,0,0,0,0,0,0,1,1,1,0,0,0],[0,1,1,0,1,0,0,0,0,0,0,0,0],[0,1,0,0,1,1,0,0,1,0,1,0,0],[0,1,0,0,1,1,0,0,1,1,1,0,0],[0,0,0,0,0,0,0,0,0,0,1,0,0],[0,0,0,0,0,0,0,1,1,1,0,0,0],[0,0,0,0,0,0,0,1,1,0,0,0,0]]

输出：6

解释：

答案不应该是 11 ，因为岛屿只能包含水平或垂直这四个方向上的 1 。

示例 2：

输入：

grid = [[0,0,0,0,0,0,0,0]]

输出：0

提示：

m == grid.length

n == grid[i].length

1 <= m, n <= 50

grid[i][j] 为 0 或 1

3. **算法思路：**

- 遍历整个矩阵，每当遇到一块土地的时候，就用「深搜」或者「宽搜」将与这块土地相连的「整个岛屿」的面积计算出来。
- 然后在搜索得到的「所有的岛屿面积」求一个「最大值」即可。
- 在搜索过程中，为了「防止搜到重复的土地」：
  - 可以开一个同等规模的「布尔数组」，标记一下这个位置是否已经被访问过；
  - 也可以将原始矩阵的 `1` 修改成 `0`，但是这样操作会修改原始矩阵。

4. **解法（深搜 dfs）：**

算法流程：

- 主函数内：
  a. 遍历整个数组，发现一块没有遍历到的土地之后，就用 `dfs`，将与这块土地相连的岛屿的面积求出来；
  b. 然后将面积更新到最终结果 `ret` 中。
- 深搜函数 `dfs` 中：
  a. 能够进到 `dfs` 函数中，说明是一个没遍历到的位置；
  b. 标记一下已经遍历过，设置一个变量 `S = 1`（当前这个位置的面积为 `1`），记录最终的面积；
  c. 上下左右遍历四个位置：
    - 如果找到一块没有遍历到的土地，就将与这块土地相连的岛屿面积累加到 `S` 上；
  d. 循环结束后，`S` 中存的就是整块岛屿的面积，返回即可。

C++ 算法代码：

```
class Solution
{
    bool vis[51][51];
    int m, n;
    int dx[4] = {0, 0, 1, -1};
    int dy[4] = {1, -1, 0, 0};
    int count;

public:
    int maxAreaOfIsland(vector<vector<int>>& grid)
    {
```

```
12          m = grid.size(), n = grid[0].size();
13
14          int ret = 0;
15          for(int i = 0; i < m; i++)
16              for(int j = 0; j < n; j++)
17                  if(!vis[i][j] && grid[i][j] == 1)
18                  {
19                      count = 0;
20                      dfs(grid, i, j);
21                      ret = max(ret, count);
22                  }
23          return ret;
24      }
25
26      void dfs(vector<vector<int>>& grid, int i, int j)
27      {
28          count++;
29          vis[i][j] = true;
30
31          for(int k = 0; k < 4; k++)
32          {
33              int x = i + dx[k], y = j + dy[k];
34              if(x >= 0 && x < m && y >= 0 && y < n && !vis[x][y] && grid[x][y]
    == 1)
35              {
36                  dfs(grid, x, y);
37              }
38          }
39      }
40  };
```

C++ 运行结果：

Java 算法代码：

```
1  class Solution
2  {
3      boolean[][] vis;
```

```java
    int m, n;
    int[] dx = {0, 0, -1, 1};
    int[] dy = {1, -1, 0, 0};
    int count;

    public int maxAreaOfIsland(int[][] grid)
    {
        m = grid.length; n = grid[0].length;
        vis = new boolean[m][n];

        int ret = 0;
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
            {
                if(!vis[i][j] && grid[i][j] == 1)
                {
                    count = 0;
                    dfs(grid, i, j);
                    ret = Math.max(ret, count);
                }
            }
        return ret;
    }

    public void dfs(int[][] grid, int i, int j)
    {
        count++;
        vis[i][j] = true;

        for(int k = 0; k < 4; k++)
        {
            int x = i + dx[k], y = j + dy[k];
            if(x >= 0 && x < m && y >= 0 && y < n && !vis[x][y] && grid[x][y] == 1)
                dfs(grid, x, y);
        }
    }
}
```


Java 运行结果：

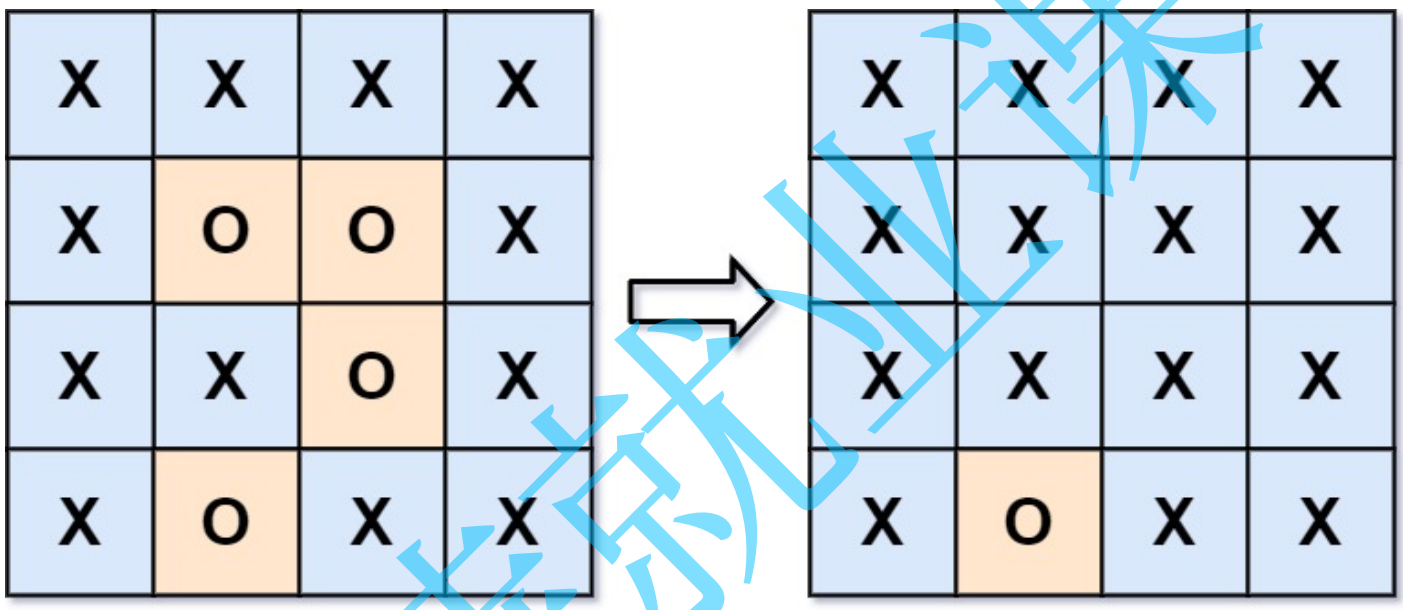
32. 被围绕的区域（medium）

1. **题目链接：130. 被围绕的区域**

2. **题目描述：**

给你一个 m x n 的矩阵 board ，由若干字符 'X' 和 'O' ，找到所有被 'X' 围绕的区域，并将这些区域里所有的 'O' 用 'X' 填充。

示例 1：



输入：board = [["X","X","X","X"],["X","O","O","X"],["X","X","O","X"],["X","O","X","X"]]

输出：[["X","X","X","X"],["X","X","X","X"],["X","X","X","X"],["X","O","X","X"]]

解释：被围绕的区间不会存在于边界上，换句话说，任何边界上的 'O' 都不会被填充为 'X'。 任何不在边界上，或不与边界上的 'O' 相连的 'O' 最终都会被填充为 'X'。如果两个元素在水平或垂直方向相邻，则称它们是“相连”的。

示例 2：

输入：board = [["X"]]

输出：[["X"]]

提示：

m == board.length

n == board[i].length

1 <= m, n <= 200

> board[i][j] 为 'X' 或 'O'

3. **解法：**

算法思路：

正难则反。

可以先利用 `dfs` 将与边缘相连的 `'0'` 区域做上标记，然后重新遍历矩阵，将没有标记过的 `'0'` 修改成 `'X'` 即可。

C++ 算法代码：

```
class Solution
{
    int dx[4] = {1, -1, 0, 0};
    int dy[4] = {0, 0, 1, -1};
    int m, n;

public:
    void solve(vector<vector<char>>& board)
    {
        m = board.size(), n = board[0].size();

        // 1. 把边界的 O 相连的联通块，全部修改成 .
        for(int j = 0; j < n; j++)
        {
            if(board[0][j] == 'O') dfs(board, 0, j);
            if(board[m - 1][j] == 'O') dfs(board, m - 1, j);
        }
        for(int i = 0; i < m; i++)
        {
            if(board[i][0] == 'O') dfs(board, i, 0);
            if(board[i][n - 1] == 'O') dfs(board, i, n - 1);
        }

        // 2. 还原
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
            {
                if(board[i][j] == '.') board[i][j] = 'O';
                else if(board[i][j] == 'O') board[i][j] = 'X';
            }
    }

    void dfs(vector<vector<char>>& board, int i, int j)
    {
```

```
        board[i][j] = '.';
        for(int k = 0; k < 4; k++)
        {
            int x = i + dx[k], y = j + dy[k];
            if(x >= 0 && x < m && y >= 0 && y < n && board[x][y] == 'O')
            {
                dfs(board, x, y);
            }
        }
    }
};
```

C++ 代码结果：

Java 算法代码：

```java
class Solution
{
    int m, n;
    int[] dx = {1, -1, 0, 0};
    int[] dy = {0, 0, 1, -1};

    public void solve(char[][] board)
    {
        m = board.length; n = board[0].length;

        // 1. 先把边界的 O 相连的联通块，全部修改成 '.'
        for(int j = 0; j < n; j++)
        {
            if(board[0][j] == 'O') dfs(board, 0, j);
            if(board[m - 1][j] == 'O') dfs(board, m - 1, j);
        }
        for(int i = 0; i < m; i++)
        {
            if(board[i][0] == 'O') dfs(board, i, 0);
            if(board[i][n - 1] == 'O') dfs(board, i, n - 1);
        }

```

```java
        // 2. 还原
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
            {
                if(board[i][j] == '.') board[i][j] = 'O';
                else if(board[i][j] == 'O') board[i][j] = 'X';
            }
    }

    public void dfs(char[][] board, int i, int j)
    {
        board[i][j] = '.';
        for(int k = 0; k < 4; k++)
        {
            int x = i + dx[k], y = j + dy[k];
            if(x >= 0 && x < m && y >= 0 && y < n && board[x][y] == 'O')
            {
                dfs(board, x, y);
            }
        }
    }
}
```

Java 运行结果：

33. 太平洋大西洋水流问题（medium）

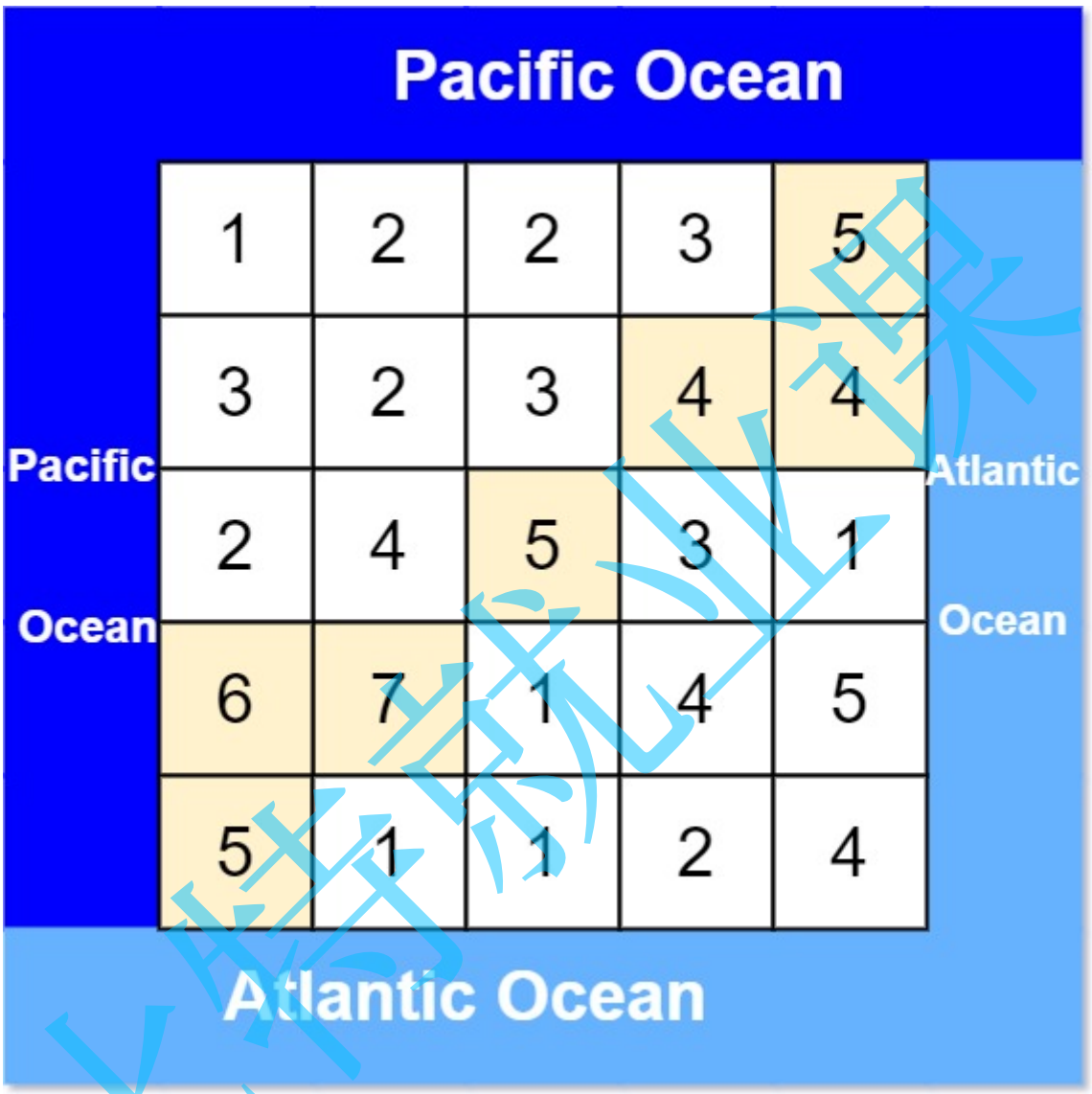
1. **题目链接：417. 太平洋大西洋水流问题**

2. **题目描述：**

> 有一个 m × n 的矩形岛屿，与 太平洋 和 大西洋 相邻。 “太平洋” 处于大陆的左边界和上边界，而 “大西洋” 处于大陆的右边界和下边界。
>
> 这个岛被分割成一个由若干方形单元格组成的网格。给定一个 m x n 的整数矩阵 heights ， heights[r][c] 表示坐标 (r, c) 上单元格 高于海平面的高度 。
>
> 岛上雨水较多，如果相邻单元格的高度 小于或等于 当前单元格的高度，雨水可以直接向北、南、东、西流向相邻单元格。水可以从海洋附近的任何单元格流入海洋。

返回网格坐标 result 的 2D 列表 ， 其中 result[i] = [ri, ci] 表示雨水从单元格 (ri, ci) 流动 既可流向太平洋也可流向大西洋 。

示例 1：



输入: heights = [[1,2,2,3,5],[3,2,3,4,4],[2,4,5,3,1],[6,7,1,4,5],[5,1,1,2,4]]

输出: [[0,4],[1,3],[1,4],[2,2],[3,0],[3,1],[4,0]]

示例 2：

输入: heights = [[2,1],[1,2]]

输出: [[0,0],[0,1],[1,0],[1,1]]

提示：

m == heights.length

> n == heights[r].length
>
> 1 <= m, n <= 200
>
> 0 <= heights[r][c] <= 10^5

3. **解法：**

算法思路：

正难则反。

如果直接去判断某一个位置是否既能到大西洋也能到太平洋，会重复遍历很多路径。

我们反着来，从大西洋沿岸开始反向 `dfs` ，这样就能找出那些点可以流向大西洋；同理，从太平洋沿岸也反向 `dfs` ，这样就能找出那些点可以流向太平洋。那么，被标记两次的点，就是我们要找的结果。

C++ 算法代码：

```
class Solution
{
    int m, n;
    int dx[4] = {0, 0, 1, -1};
    int dy[4] = {1, -1, 0, 0};

public:
    vector<vector<int>> pacificAtlantic(vector<vector<int>>& h)
    {
        m = h.size(), n = h[0].size();
        vector<vector<bool>> pac(m, vector<bool>(n));
        vector<vector<bool>> atl(m, vector<bool>(n));

        // 1. 先处理 pac 洋
        for(int j = 0; j < n; j++) dfs(h, 0, j, pac);
        for(int i = 0; i < m; i++) dfs(h, i, 0, pac);

        // 2. 处理 atl 洋
        for(int i = 0; i < m; i++) dfs(h, i, n - 1, atl);
        for(int j = 0; j < n; j++) dfs(h, m - 1, j, atl);

        vector<vector<int>> ret;
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
                if(pac[i][j] && atl[i][j])
                    ret.push_back({i, j});
        return ret;
    }
```

```
29
30      void dfs(vector<vector<int>>& h, int i, int j, vector<vector<bool>>& vis)
31      {
32          vis[i][j] = true;
33          for(int k = 0; k < 4; k++)
34          {
35              int x = i + dx[k], y = j + dy[k];
36              if(x >= 0 && x < m && y >= 0 && y < n && !vis[x][y] && h[x][y] >=
    h[i][j])
37                  dfs(h, x, y, vis);
38          }
39      }
40  };
```

C++ 代码结果：

Java 算法代码：

```java
class Solution
{
    int m, n;
    int[] dx = {0, 0, 1, -1};
    int[] dy = {1, -1, 0, 0};

    public List<List<Integer>> pacificAtlantic(int[][] h)
    {
        m = h.length; n = h[0].length;

        boolean[][] pac = new boolean[m][n];
        boolean[][] atl = new boolean[m][n];

        // 1. 先处理 pac 洋
        for(int j = 0; j < n; j++) dfs(h, 0, j, pac);
        for(int i = 0; i < m; i++) dfs(h, i, 0, pac);

        // 2. 再处理 atl 洋
        for(int i = 0; i < m; i++) dfs(h, i, n - 1, atl);
        for(int j = 0; j < n; j++) dfs(h, m - 1, j, atl);
```

```java

        // 3. 提取结果
        List<List<Integer>> ret = new ArrayList<>();
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
                if(pac[i][j] && atl[i][j])
                {
                    List<Integer> tmp = new ArrayList<>();
                    tmp.add(i); tmp.add(j);
                    ret.add(tmp);
                }

        return ret;
    }

    public void dfs(int[][] h, int i, int j, boolean[][] vis)
    {
        vis[i][j] = true;
        for(int k = 0; k < 4; k++)
        {
            int x = i + dx[k], y = j + dy[k];
            if(x >= 0 && x < m && y >= 0 && y < n && !vis[x][y] && h[x][y] >= h[i][j])
            {
                dfs(h, x, y, vis);
            }
        }
    }
}
```

Java 运行结果：

34. 扫雷游戏（medium）

1. **题目链接：529. 扫雷游戏**

2. **题目描述：**

> 让我们一起来玩扫雷游戏！

给你一个大小为 m x n 二维字符矩阵 board ，表示扫雷游戏的盘面，其中：

'M' 代表一个 未挖出的 地雷，

'E' 代表一个 未挖出的 空方块，

'B' 代表没有相邻（上，下，左，右，和所有4个对角线）地雷的 已挖出的 空白方块，

数字（'1' 到 '8'）表示有多少地雷与这块 已挖出的 方块相邻，

'X' 则表示一个 已挖出的 地雷。

给你一个整数数组 click ，其中 click = [clickr, clickc] 表示在所有 未挖出的 方块（'M' 或者 'E'）中的下一个点击位置（clickr 是行下标，clickc 是列下标）。

根据以下规则，返回相应位置被点击后对应的盘面：

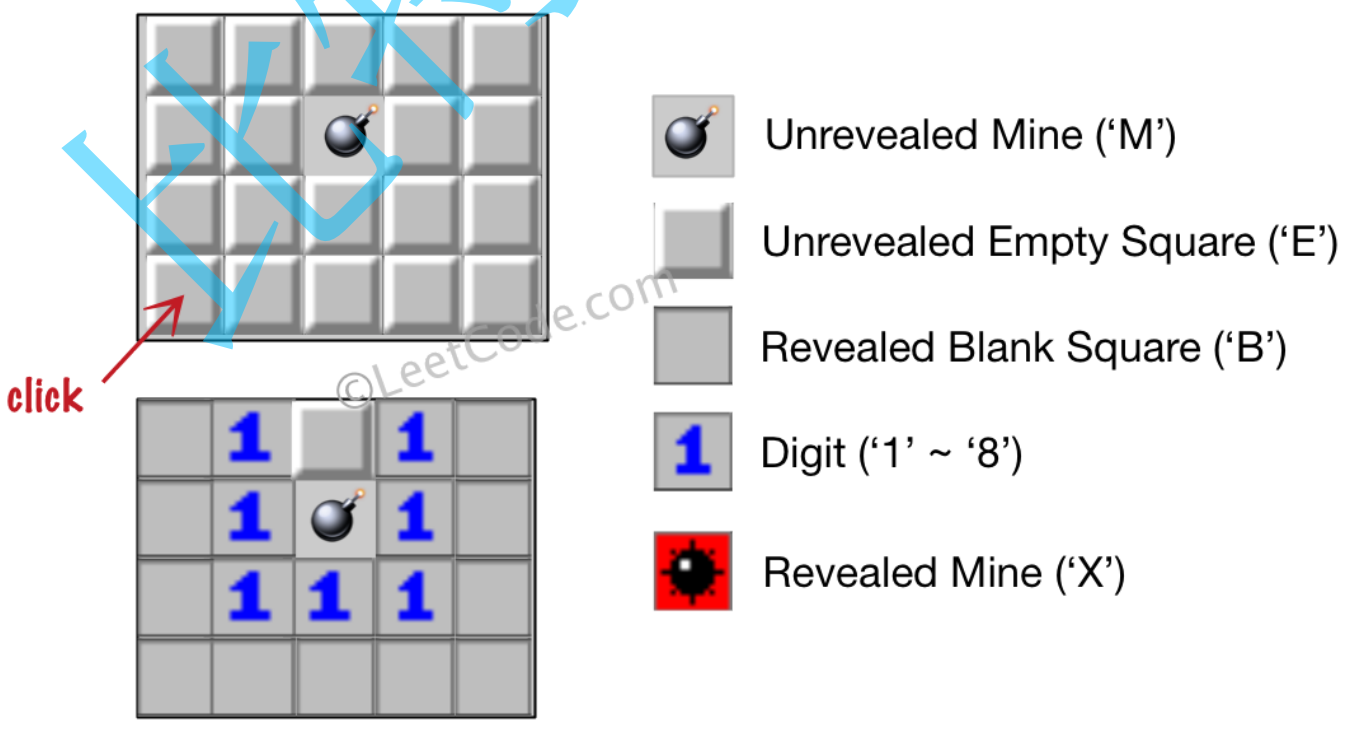
如果一个地雷（'M'）被挖出，游戏就结束了- 把它改为 'X'。

如果一个 没有相邻地雷 的空方块（'E'）被挖出，修改它为（'B'），并且所有和其相邻的 未挖出 方块都应该被递归地揭露。

如果一个 至少与一个地雷相邻 的空方块（'E'）被挖出，修改它为数字（'1' 到 '8'），表示相邻地雷的数量。

如果在此次点击中，若无更多方块可被揭露，则返回盘面。

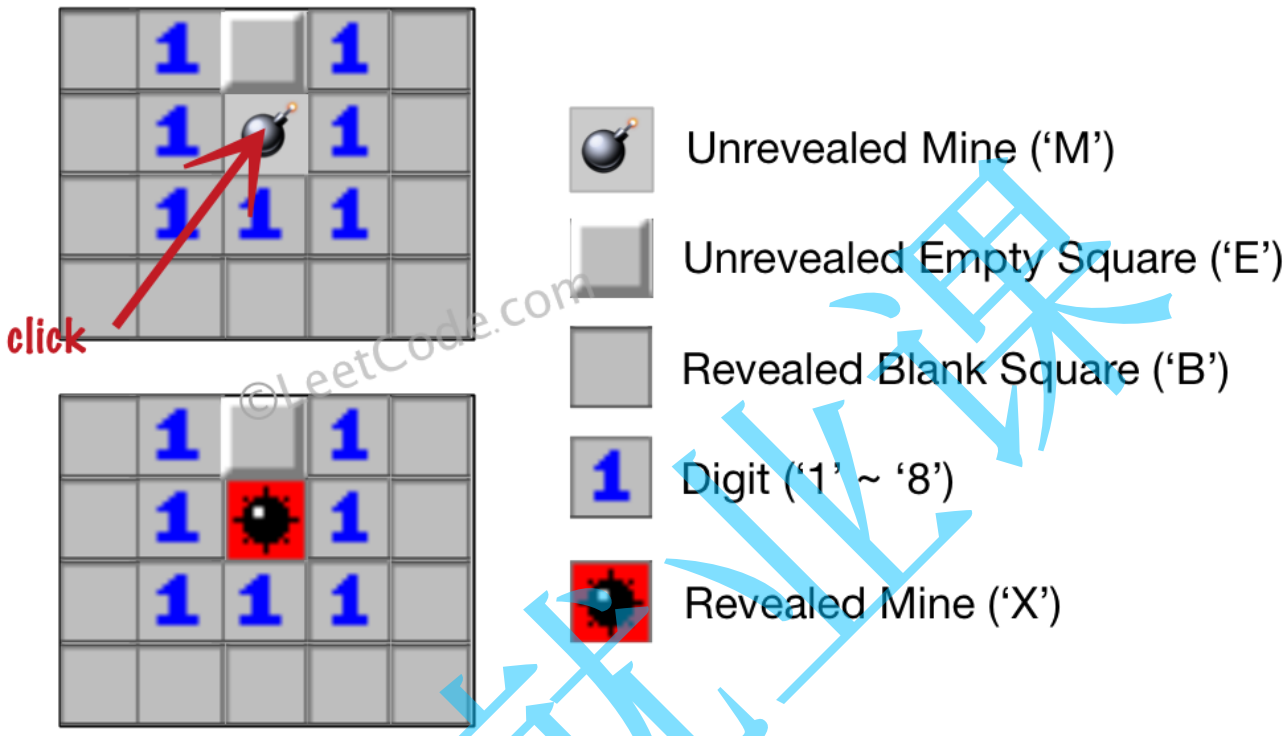
示例 1：



输入：board = [["E","E","E","E","E"],["E","E","M","E","E"],["E","E","E","E","E"],["E","E","E","E","E"]], click = [3,0]

输出：[["B","1","E","1","B"],["B","1","M","1","B"],["B","1","1","1","B"],["B","B","B","B","B"]]

示例 2：



输入：board = [["B","1","E","1","B"],["B","1","M","1","B"],["B","1","1","1","B"],["B","B","B","B","B"]], click = [1,2]

输出：[["B","1","E","1","B"],["B","1","X","1","B"],["B","1","1","1","B"],["B","B","B","B","B"]]

提示：

m == board.length

n == board[i].length

1 <= m, n <= 50

board[i][j] 为 'M'、'E'、'B' 或数字 '1' 到 '8' 中的一个

click.length == 2

0 <= clickr < m

0 <= clickc < n

board[clickr][clickc] 为 'M' 或 'E'

3. **解法：**

算法思路：

模拟类型的 `dfs` 题目。

首先要搞懂题目要求，也就是游戏规则。

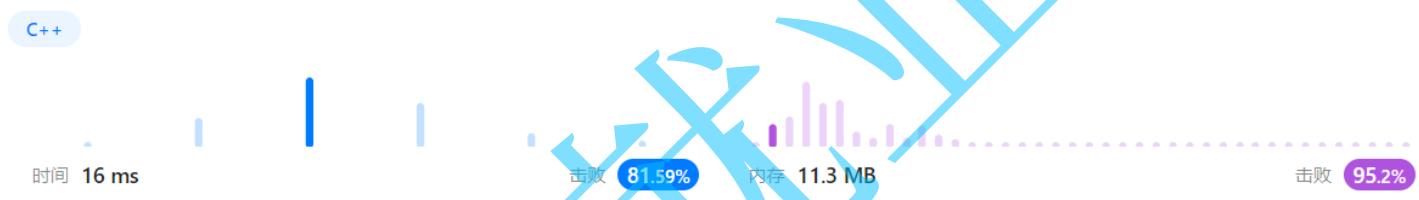
从题目所给的点击位置开始，根据游戏规则，来一次 `dfs` 即可。

C++ 算法代码：

```
class Solution
{
    int dx[8] = {0, 0, 1, -1, 1, 1, -1, -1};
    int dy[8] = {1, -1, 0, 0, 1, -1, 1, -1};
    int m, n;

public:
    vector<vector<char>> updateBoard(vector<vector<char>>& board, vector<int>& click)
    {
        m = board.size(), n = board[0].size();
        int x = click[0], y = click[1];
        if(board[x][y] == 'M') // 直接点到地雷
        {
            board[x][y] = 'X';
            return board;
        }
        dfs(board, x, y);
        return board;
    }

    void dfs(vector<vector<char>>& board, int i, int j)
    {
        // 统计一下周围的地雷个数
        int count = 0;
        for(int k = 0; k < 8; k++)
        {
            int x = i + dx[k], y = j + dy[k];
            if(x >= 0 && x < m && y >= 0 && y < n && board[x][y] == 'M')
            {
                count++;
            }
        }

        if(count) // 周围有地雷
        {
            board[i][j] = count + '0';
            return;
```

```
        }
        else // 周围没有地雷
        {
            board[i][j] = 'B';
            for(int k = 0; k < 8; k++)
            {
                int x = i + dx[k], y = j + dy[k];
                if(x >= 0 && x < m && y >= 0 && y < n && board[x][y] == 'E')
                {
                   dfs(board, x, y);
                }
            }
        }
    }
};
```

C++ 代码结果：



Java 算法代码：

```
class Solution
{
    int[] dx = {0, 0, 1, -1, 1, 1, -1, -1};
    int[] dy = {1, -1, 0, 0, 1, -1, 1, -1};
    int m, n;

    public char[][] updateBoard(char[][] board, int[] click)
    {
        m = board.length; n = board[0].length;
        int x = click[0], y = click[1];

        if(board[x][y] == 'M') // 如果直接点到地雷，游戏结束
        {
            board[x][y] = 'X';
            return board;
        }

        dfs(board, x, y);
```

```
19          return board;
20      }
21
22      public void dfs(char[][] board, int i, int j)
23      {
24          // 统计一下周围地雷的个数
25          int count = 0;
26          for(int k = 0; k < 8; k++)
27          {
28              int x = i + dx[k], y = j + dy[k];
29              if(x >= 0 && x < m && y >= 0 && y < n && board[x][y] == 'M')
30              {
31                  count++;
32              }
33          }
34
35          if(count == 0) // 周围没有地雷
36          {
37              board[i][j] = 'B';
38              for(int k = 0; k < 8; k++)
39              {
40                  int x = i + dx[k], y = j + dy[k];
41                  if(x >= 0 && x < m && y >= 0 && y < n && board[x][y] == 'E')
42                  {
43                      dfs(board, x, y);
44                  }
45              }
46          }
47          else
48          {
49              board[i][j] = (char)('0' + count);
50              return;
51          }
52      }
53  }
```

Java 运行结果：

35. 机器人的运动范围（medium）

1. **题目链接：面试题 13. 机器人的运动范围**

2. **题目描述：**

> 地上有一个 m 行 n 列的方格，从坐标 [0,0] 到坐标 [m-1,n-1] 。一个机器人从坐标 [0, 0] 的格子开始移动，它每次可以向左、右、上、下移动一格（不能移动到方格外），也不能进入行坐标和列坐标的数位之和大于 k 的格子。例如，当 k 为 18 时，机器人能够进入方格 [35, 37] ，因为 3+5+3+7=18。但它不能进入方格 [35, 38]，因为 3+5+3+8=19。请问该机器人能够到达多少个格子？
>
> 示例 1：
>
> 输入：m = 2, n = 3, k = 1
>
> 输出：3
>
> 示例 2：
>
> 输入：m = 3, n = 1, k = 0
>
> 输出：1
>
> 提示：
>
> - `1 <= n,m <= 100`
> - `0 <= k <= 20`

3. **算法思路：**

这是一道非常典型的「搜索」类问题。

我们可以通过「深搜」或者「宽搜」，从 `[0, 0]` 点出发，按照题目的「规则」一直往 `[m - 1, n - 1]` 位置走。

同时，设置一个全局变量。每次走到一个合法位置，就将全局变量加一。当我们把所有能走到的路都走完之后，全局变量里面存的就是最终答案。

4. **解法（dfs）：**

算法流程：

- **递归函数设计：**
  a. 参数：当前所在的位置 `[i, j]`，行走的边界 `[m, n]`，坐标数位之和的边界 `k`；
  b. 递归出口：
    i. `[i, j]` 的坐标不合法，也就是已经超出能走的范围；

ii. `[i, j]` 位置已经走过了（因此我们需要创建一个全局变量 `bool st[101][101]`，来标记当前位置是否走过）；

iii. `[i, j]` 坐标的数位之和大于 `k`；

上述情况的任何一种都是递归出口。

c. 函数体内部：

i. 如果这个坐标是合法的，就将全局变量 `ret++`；

ii. 然后标记一下 `[i, j]` 位置已经遍历过；

iii. 然后去 `[i, j]` 位置的上下左右四个方向去看看。

- **辅助函数：**

a. 检测坐标 `[i, j]` 是否合法；

b. 计算出 `i, j` 的数位之和，然后与 `k` 作比较即可。

- **主函数：**

a. 调用递归函数，从 `[0 ,0]` 点出发。

- **辅助的全局变量：**

a. 二维数组 `bool st[101][101]`：标记 `[i, j]` 位置是否已经遍历过；

b. 变量 `ret` ：记录一共到达多少个合法的位置。

c. 上下左右的四个坐标变换。

C++ 算法代码：

```cpp
class Solution
{
    int m, n, k;
    bool vis[101][101];
    int ret;
    int dx[4] = {0, 0, -1, 1};
    int dy[4] = {1, -1, 0, 0};

public:
    int movingCount(int _m, int _n, int _k)
    {
        m = _m, n = _n, k = _k;
        dfs(0, 0);
        return ret;
    }

```

```
    void dfs(int i, int j)
    {
        ret++;
        vis[i][j] = true;

        for(int k = 0; k < 4; k++)
        {
            int x = i + dx[k], y = j + dy[k];
            if(x >= 0 && x < m && y >= 0 && y < n && !vis[x][y] && check(x, y))
                dfs(x, y);
        }
    }

    bool check(int i, int j)
    {
        int tmp = 0;
        while(i)
        {
            tmp += i % 10;
            i /= 10;
        }
        while(j)
        {
            tmp += j % 10;
            j /= 10;
        }
        return tmp <= k;
    }
};
```

C++ 运行结果：

Java 算法代码：

```
class Solution
{
    int m, n, k;
    boolean[][] vis;
```

```java
    int[] dx = {1, -1, 0, 0};
    int[] dy = {0, 0, 1, -1};
    int ret;

    public int movingCount(int _m, int _n, int _k)
    {
        m = _m; n = _n; k = _k;
        vis = new boolean[m][n];
        dfs(0, 0);
        return ret;
    }

    public void dfs(int i, int j)
    {
        ret++;
        vis[i][j] = true;

        for(int k = 0; k < 4; k++)
        {
            int x = i + dx[k], y = j + dy[k];
            if(x >= 0 && x < m && y >= 0 && y < n && !vis[x][y] && check(x, y))
            {
                dfs(x, y);
            }
        }
    }

    public boolean check(int i, int j)
    {
        int tmp = 0;
        while(i != 0)
        {
            tmp += i % 10;
            i /= 10;
        }
        while(j != 0)
        {
            tmp += j % 10;
            j /= 10;
        }
        return tmp <= k;
    }
}
```


Java 运行结果：

记忆化搜索

36. 斐波那契数（easy）

1. **题目链接：509. 斐波那契数**

2. **题目描述：**

斐波那契数 （通常用 F(n) 表示）形成的序列称为 斐波那契数列 。该数列由 0 和 1 开始，后面的每一项数字都是前面两项数字的和。也就是：

F(0) = 0，F(1) = 1

F(n) = F(n - 1) + F(n - 2)，其中 n > 1

给定 n ，请计算 F(n) 。

示例 1：

输入：n = 2

输出：1

解释：F(2) = F(1) + F(0) = 1 + 0 = 1

示例 2：

输入：n = 3

输出：2

解释：F(3) = F(2) + F(1) = 1 + 1 = 2

示例 3：

输入：n = 4

输出：3

解释：F(4) = F(3) + F(2) = 2 + 1 = 3

提示：

> 0 <= n <= 30

3. **解法（暴搜 -> 记忆化搜索 -> 动态规划）：**

算法思路：

暴搜：

a. 递归含义：给 `dfs` 一个使命，给他一个数 `n`，返回第 `n` 个斐波那契数的值；

b. 函数体：斐波那契数的递推公式；

c. 递归出口：当 `n == 0` 或者 `n == 1` 时，不用套公式。

记忆化搜索：

a. 加上一个备忘录；

b. 每次进入递归的时候，去备忘录里面看看；

c. 每次返回的时候，将结果加入到备忘录里面。

动态规划：

a. 递归含义 -> 状态表示；

b. 函数体 -> 状态转移方程；

c. 递归出口 -> 初始化。

C 算法代码：

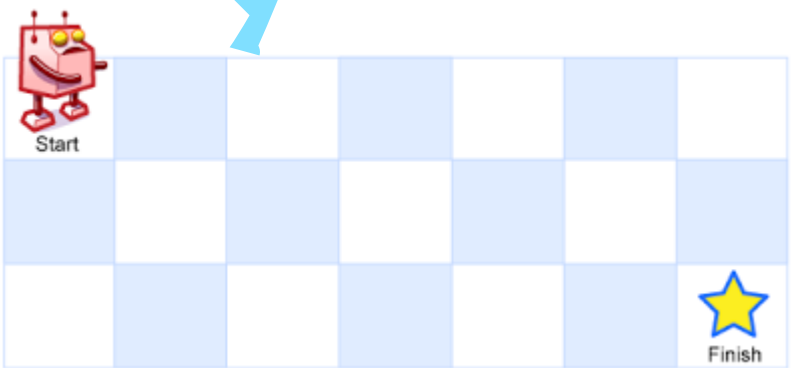
```c
int memo[31]; // memory 备忘录

int dp[31];
int fib(int n)
{
    // 动态规划
    dp[0] = 0; dp[1] = 1;
    for(int i = 2; i <= n; i++)
        dp[i] = dp[i - 1] + dp[i - 2];
    return dp[n];
}

// 记忆化搜索
int dfs(int n)
{
    if(memo[n] != -1)
```

```
17      {
18          return memo[n]; // 直接去备忘录里面拿值
19      }
20
21      if(n == 0 || n == 1)
22      {
23          memo[n] = n; // 记录到备忘录里面
24          return n;
25      }
26      memo[n] = dfs(n - 1) + dfs(n - 2); // 记录到备忘录里面
27      return memo[n];
28  }
```

C 运行结果：

37. 不同路径（medium）

1. **题目链接：62. 不同路径**
2. **题目描述：**

> 一个机器人位于一个 m x n 网格的左上角（起始点在下图中标记为 “Start” ）。
>
> 机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为 “Finish” ）。
>
> 问总共有多少条不同的路径?
>
> **示例 1：**
>
> 
>
>
> 输入：m = 3, n = 7
>
> 输出：28

> 示例 2：
>
> 输入：m = 3, n = 2
>
> 输出：3
>
> 解释：
>
> 从左上角开始，总共有 3 条路径可以到达右下角。
>
> 1. 向右 -> 向下 -> 向下
> 2. 向下 -> 向下 -> 向右
> 3. 向下 -> 向右 -> 向下

3. **解法（暴搜 -> 记忆化搜索 -> 动态规划）：**

算法思路：

暴搜：

a. 递归含义：给 `dfs` 一个使命，给他一个下标，返回从 `[0, 0]` 位置走到 `[i, j]` 位置一共有多少种方法；

b. 函数体：只要知道到达上面位置的方法数以及到达左边位置的方法数，然后累加起来即可；

c. 递归出口：当下标越界的时候返回 `0`；当位于起点的时候，返回 `1`。

记忆化搜索：

a. 加上一个备忘录；

b. 每次进入递归的时候，去备忘录里面看看；

c. 每次返回的时候，将结果加入到备忘录里面。

动态规划：

a. 递归含义 -> 状态表示；

b. 函数体 -> 状态转移方程；

c. 递归出口 -> 初始化。

C++ 算法代码：

```
class Solution
{
public:
    int uniquePaths(int m, int n)
```

```cpp
    {
        // 动态规划
        vector<vector<int>> dp(m + 1, vector<int>(n + 1));
        dp[1][1] = 1;
        for(int i = 1; i <= m; i++)
            for(int j = 1; j <= n; j++)
            {
                if(i == 1 && j == 1) continue;
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
            }
        return dp[m][n];

        // 记忆化搜索
        // vector<vector<int>> memo(m + 1, vector<int>(n + 1));
        // return dfs(m, n, memo);
    }

    int dfs(int i, int j, vector<vector<int>>& memo)
    {
        if(memo[i][j] != 0)
        {
            return memo[i][j];
        }

        if(i == 0 || j == 0) return 0;
        if(i == 1 && j == 1)
        {
            memo[i][j] = 1;
            return 1;
        }

        memo[i][j] = dfs(i - 1, j, memo) + dfs(i, j - 1, memo);
        return memo[i][j];
    }
};
```


C++ 运行结果：

Java 算法代码：

```java
class Solution
{
    public int uniquePaths(int m, int n)
    {
        // 动态规划
        int[][] dp = new int[m + 1][n + 1];
        dp[1][1] = 1;
        for(int i = 1; i <= m; i++)
            for(int j = 1; j <= n; j++)
            {
                if(i == 1 && j == 1) continue;
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
            }
        return dp[m][n];

        // 记忆化搜索
        // int[][] memo = new int[m + 1][n + 1];
        // return dfs(m, n, memo);
    }

    public int dfs(int i, int j, int[][] memo)
    {
        if(memo[i][j] != 0)
        {
            return memo[i][j];
        }

        if(i == 0 || j == 0) return 0;
        if(i == 1 && j == 1)
        {
            memo[i][j] = 1;
            return 1;
        }

        memo[i][j] = dfs(i - 1, j, memo) + dfs(i, j - 1, memo);
        return memo[i][j];
    }
}
```


Java 运行结果：

38. 最长递增子序列（medium）

1. **题目链接：300. 最长递增子序列**

2. **题目描述：**

> 给你一个整数数组 nums ，找到其中最长严格递增子序列的长度。
>
> 子序列 是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，[3,6,2,7] 是数组 [0,3,1,6,2,2,7] 的子序列。
>
> 示例 1：
>
> 输入：nums = [10,9,2,5,3,7,101,18]
>
> 输出：4
>
> 解释：最长递增子序列是 [2,3,7,101]，因此长度为 4 。
>
> 示例 2：
>
> 输入：nums = [0,1,0,3,2,3]
>
> 输出：4
>
> 示例 3：
>
> 输入：nums = [7,7,7,7,7,7,7]
>
> 输出：1
>
> 提示：
>
> 1 <= nums.length <= 2500
>
> -10^4 <= nums[i] <= 10^4

3. **解法（暴搜 -> 记忆化搜索 -> 动态规划）：**

算法思路：

暴搜：

a. 递归含义：给 `dfs` 一个使命，给他一个数 `i` ，返回以 `i` 位置为起点的最长递增子序列的长度；

b. 函数体：遍历 `i` 后面的所有位置，看看谁能加到 `i` 这个元素的后面。统计所有情况下的最大值。

c. 递归出口：因为我们是判断之后再进入递归的，因此没有出口~

记忆化搜索：

a. 加上一个备忘录；

b. 每次进入递归的时候，去备忘录里面看看；

c. 每次返回的时候，将结果加入到备忘录里面。

动态规划：

a. 递归含义 -> 状态表示；

b. 函数体 -> 状态转移方程；

c. 递归出口 -> 初始化。

C++ 算法代码：

```cpp
class Solution
{
public:
    int lengthOfLIS(vector<int>& nums)
    {
        // 动态规划
        int n = nums.size();
        vector<int> dp(n, 1);
        int ret = 0;
        // 填表顺序：从后往前
        for(int i = n - 1; i >= 0; i--)
        {
            for(int j = i + 1; j < n; j++)
            {
                if(nums[j] > nums[i])
                {
                    dp[i] = max(dp[i], dp[j] + 1);
                }
            }
            ret = max(ret, dp[i]);
        }
        return ret;

```

```
24
25          // 记忆化搜索
26          //
27          // vector<int> memo(n);
28
29          // int ret = 0;
30          // for(int i = 0; i < n; i++)
31          //     ret = max(ret, dfs(i, nums, memo));
32          // return ret;
33      }
34
35      int dfs(int pos, vector<int>& nums, vector<int>& memo)
36      {
37          if(memo[pos] != 0) return memo[pos];
38
39          int ret = 1;
40          for(int i = pos + 1; i < nums.size(); i++)
41          {
42              if(nums[i] > nums[pos])
43              {
44                  ret = max(ret, dfs(i, nums, memo) + 1);
45              }
46          }
47          memo[pos] = ret;
48          return ret;
49      }
50  };
```

C++ 代码结果：

Java 算法代码：

```
1  class Solution
2  {
3      public int lengthOfLIS(int[] nums)
4      {
5          int n = nums.length;
6          int[] dp = new int[n];
```

```java
        int ret = 0;
        Arrays.fill(dp, 1);
        // 填表顺序：从后往前
        for(int i = n - 1; i >= 0; i--)
        {
            for(int j = i + 1; j < n; j++)
            {
                if(nums[j] > nums[i])
                {
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                }
            }
            ret = Math.max(ret, dp[i]);
        }
        return ret;


        // 记忆化搜索
        // int ret = 0;
        // int n = nums.length;
        // int[] memo = new int[n];

        // for(int i = 0; i < n; i++)
        // {
        //     ret = Math.max(ret, dfs(i, nums, memo));
        // }
        // return ret;
    }

    public int dfs(int pos, int[] nums, int[] memo)
    {
        if(memo[pos] != 0) return memo[pos];

        int ret = 1;
        for(int i = pos + 1; i < nums.length; i++)
        {
            if(nums[i] > nums[pos])
            {
                ret = Math.max(ret, dfs(i, nums, memo) + 1);
            }
        }
        memo[pos] = ret;
        return ret;
    }
}
```

Java 运行结果：

39. 猜数字大小II（medium）

1. **题目链接：375. 猜数字大小 II**

2. **题目描述：**

我们正在玩一个猜数游戏，游戏规则如下：

a. 我从 1 到 n 之间选择一个数字。

b. 你来猜我选了哪个数字。

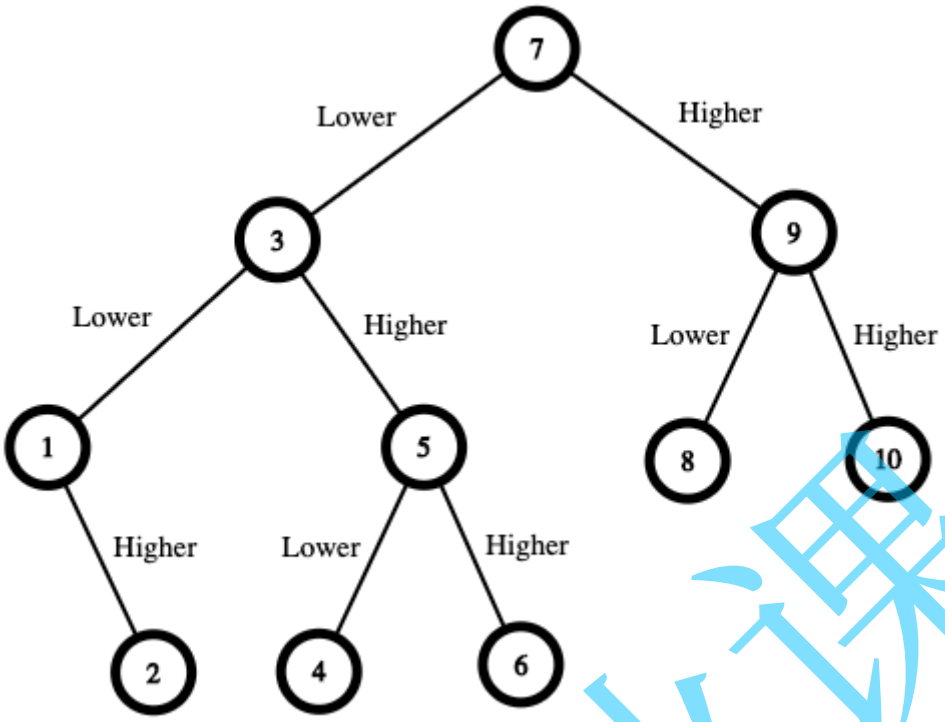
c. 如果你猜到正确的数字，就会 赢得游戏 。

d. 如果你猜错了，那么我会告诉你，我选的数字比你的 更大或者更小 ，并且你需要继续猜数。

e. 每当你猜了数字 x 并且猜错了的时候，你需要支付金额为 x 的现金。如果你花光了钱，就会输掉游戏 。

给你一个特定的数字 n ，返回能够 确保你获胜 的最小现金数，不管我选择那个数字 。

示例 1：



输入：n = 10

输出：16

解释：制胜策略如下：

- 数字范围是 [1,10] 。你先猜测数字为 7 。

- 如果这是我选中的数字，你的总费用为 $0 。否则，你需要支付 $7 。

- 如果我的数字更大，则下一步需要猜测的数字范围是 [8,10] 。你可以猜测数字为 9 。

- 如果这是我选中的数字，你的总费用为 $7 。否则，你需要支付 $9 。

- 如果我的数字更大，那么这个数字一定是 10 。你猜测数字为 10 并赢得游戏，总费用为 $7 + $9 = $16 。

- 如果我的数字更小，那么这个数字一定是 8 。你猜测数字为 8 并赢得游戏，总费用为 $7 + $9 = $16 。

- 如果我的数字更小，则下一步需要猜测的数字范围是 [1,6] 。你可以猜测数字为 3 。

- 如果这是我选中的数字，你的总费用为 $7 。否则，你需要支付 $3 。

- 如果我的数字更大，则下一步需要猜测的数字范围是 [4,6] 。你可以猜测数字为 5 。

- 如果这是我选中的数字，你的总费用为 $7 + $3 = $10 。否则，你需要支付 $5 。

- 如果我的数字更大，那么这个数字一定是 6 。你猜测数字为 6 并赢得游戏，总费用为 $7 + $3 + $5 = $15 。

- 如果我的数字更小，那么这个数字一定是 4 。你猜测数字为 4 并赢得游戏，总费用为 $7 + $3 + $5 = $15 。

- 如果我的数字更小，则下一步需要猜测的数字范围是 [1,2] 。你可以猜测数字为 1 。

- 如果这是我选中的数字，你的总费用为 $7 + $3 = $10 。否则，你需要支付 $1 。

- 如果我的数字更大，那么这个数字一定是 2 。你猜测数字为 2 并赢得游戏，总费用为 $7 + $3 + $1 = $11 。

在最糟糕的情况下，你需要支付 $16 。因此，你只需要 $16 就可以确保自己赢得游戏。

示例 2：

输入：n = 1

输出：0

解释：只有一个可能的数字，所以你可以直接猜 1 并赢得游戏，无需支付任何费用。

示例 3：

输入：n = 2

输出：1

解释：有两个可能的数字 1 和 2 。

- 你可以先猜 1 。

- 如果这是我选中的数字，你的总费用为 $0 。否则，你需要支付 $1 。

- 如果我的数字更大，那么这个数字一定是 2 。你猜测数字为 2 并赢得游戏，总费用为 $1 。

最糟糕的情况下，你需要支付 $1 。

提示：

1 <= n <= 200

3. **解法（暴搜 -> 记忆化搜索）：**

算法思路：

暴搜：

a. 递归含义：给 `dfs` 一个使命，给他一个区间 `[left, right]`，返回在这个区间上能完胜的最小费用；

b. 函数体：选择 `[left, right]` 区间上的任意一个数作为头结点，然后递归分析左右子树。求出所有情况下的最小值；

c. 递归出口：当 `left >= right` 的时候，直接返回 `0`。

记忆化搜索：

a. 加上一个备忘录；

b. 每次进入递归的时候，去备忘录里面看看；

c. 每次返回的时候，将结果加入到备忘录里面。

C++ 算法代码：

```cpp
class Solution
{
    int memo[201][201];

public:
    int getMoneyAmount(int n)
    {
        return dfs(1, n);
    }

    int dfs(int left, int right)
    {
        if(left >= right) return 0;
        if(memo[left][right] != 0) return memo[left][right];

        int ret = INT_MAX;
        for(int head = left; head <= right; head++) // 选择头结点
        {
            int x = dfs(left, head - 1);
            int y = dfs(head + 1, right);
            ret = min(ret, head + max(x, y));
        }
        memo[left][right] = ret;
        return ret;
    }
};
```

C++ 代码结果：

Java 算法代码：

```java
class Solution
{
    int[][] memo;

    public int getMoneyAmount(int n)
    {
        memo = new int[n + 1][n + 1];
        return dfs(1, n);
    }

    public int dfs(int left, int right)
    {
        if(left >= right) return 0;

        if(memo[left][right] != 0)
        {
            return memo[left][right];
        }

        int ret = Integer.MAX_VALUE;
        for(int head = left; head <= right; head++)
        {
            int x = dfs(left, head - 1);
            int y = dfs(head + 1, right);
            ret = Math.min(Math.max(x, y) + head, ret);
        }
        memo[left][right] = ret;
        return ret;
    }
}
```

Java 运行结果：

40. 矩阵中的最长递增路径（hard）

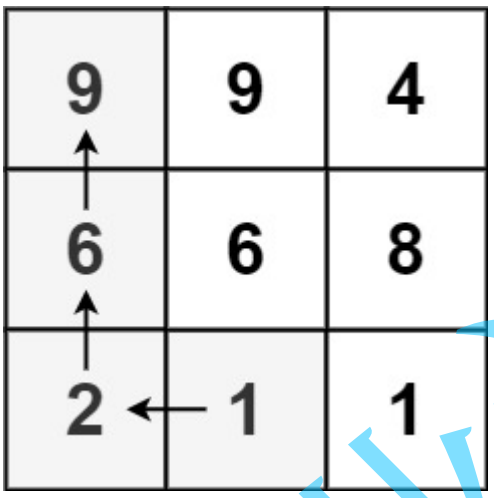
1. **题目链接：329. 矩阵中的最长递增路径**

2. **题目描述：**

> 给定一个 m x n 整数矩阵 matrix ，找出其中 最长递增路径 的长度。

对于每个单元格，你可以往上，下，左，右四个方向移动。 你 不能 在 对角线 方向上移动或移动到边界外（即不允许环绕）。

示例 1：

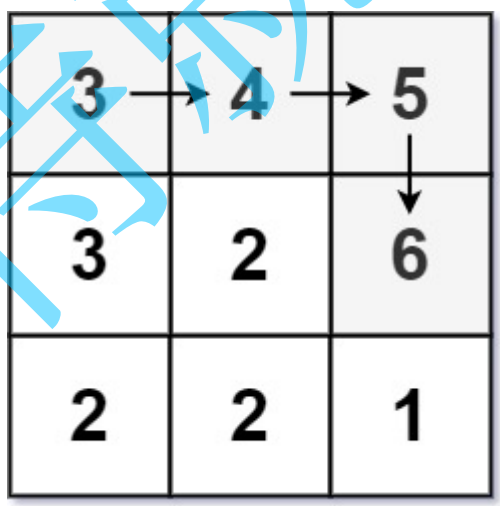


输入：matrix = [[9,9,4],[6,6,8],[2,1,1]]

输出：4

解释：最长递增路径为 [1, 2, 6, 9]。

示例 2：



输入：matrix = [[3,4,5],[3,2,6],[2,2,1]]

输出：4

解释：最长递增路径是 [3, 4, 5, 6]。注意不允许在对角线方向上移动。

示例 3：

输入：matrix = [[1]]

输出：1

> 提示：
>
> m == matrix.length
>
> n == matrix[i].length
>
> 1 <= m, n <= 200
>
> 0 <= matrix[i][j] <= 2^31 - 1

3. **解法（暴搜 -> 记忆化搜索）：**

算法思路：

暴搜：

a. 递归含义：给 `dfs` 一个使命，给他一个下标 `[i, j]`，返回从这个位置开始的最长递增路径的长度；

b. 函数体：上下左右四个方向瞅一瞅，哪里能过去就过去，统计四个方向上的最大长度；

c. 递归出口：因为我们是先判断再进入递归，因此没有出口~

记忆化搜索：

a. 加上一个备忘录；

b. 每次进入递归的时候，去备忘录里面看看；

c. 每次返回的时候，将结果加入到备忘录里面。

C++ 算法代码：

```
class Solution
{
    int m, n;
    int dx[4] = {0, 0, 1, -1};
    int dy[4] = {1, -1, 0, 0};
    int memo[201][201];

public:
    int longestIncreasingPath(vector<vector<int>>& matrix)
    {
        int ret = 0;
        m = matrix.size(), n = matrix[0].size();

        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
            {
                ret = max(ret, dfs(matrix, i, j));
```

```
            }

        return ret;
    }

    int dfs(vector<vector<int>>& matrix, int i, int j)
    {
        if(memo[i][j] != 0) return memo[i][j];

        int ret = 1;
        for(int k = 0; k < 4; k++)
        {
            int x = i + dx[k], y = j + dy[k];
            if(x >= 0 && x < m && y >= 0 && y < n && matrix[x][y] > matrix[i][j])
            {
                ret = max(ret, dfs(matrix, x, y) + 1);
            }
        }
        memo[i][j] = ret;
        return ret;
    }
};
```

C++ 代码结果：

Java 算法代码：

```
class Solution
{
    int m, n;
    int[] dx = {0, 0, 1, -1};
    int[] dy = {1, -1, 0, 0};
    int[][] memo;

    public int longestIncreasingPath(int[][] matrix)
    {
        m = matrix.length; n = matrix[0].length;
```

```java
        memo = new int[m][n];

        int ret = 0;
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
                ret = Math.max(ret, dfs(matrix, i, j));

        return ret;
    }

    public int dfs(int[][] matrix, int i, int j)
    {
        if(memo[i][j] != 0)
        {
            return memo[i][j];
        }

        int ret = 1;
        for(int k = 0; k < 4; k++)
        {
            int x = i + dx[k], y = j + dy[k];
            if(x >= 0 && x < m && y >= 0 && y < n && matrix[x][y] > matrix[i][j])
            {
                ret = Math.max(ret, dfs(matrix, x, y) + 1);
            }
        }
        memo[i][j] = ret;
        return ret;
    }
}
```

Java 运行结果：