第十一章 贪心算法讲解

11.1 贪心概念和思想

贪心算法（Greedy）是指，在对问题求解时，总是做出在**当前看来是最好**的选择。也就是说，不从整体最优上加以考虑，它所做出的仅仅是在某种意义上的**局部最优解**。

贪心算法设计的关键是贪心策略的选择。必须注意的是，贪心算法不是对所有问题都能得到整体最优解，选择的贪心策略必须具备无后效性。

11.1.1 基本思路

1. 建立数学模型来描述问题。
2. 一般结合分治的思想，把求解的问题分成若干个子问题。
3. 对每一子问题求解，得到子问题的局部最优解。
4. 把子问题的解，也就是局部最优解，合成原来解问题的一个解（可能并非全局最优，需要无后效性）。

11.1.2 实现框架

```
从问题的某一初始解出发；
  while （能朝给定总目标前进一步）
  {
        利用可行的决策，求出可行解的一个解元素；
  }
由所有解元素组合成问题的一个可行解；
```

11.1.3 适用场景

贪心算法存在的问题：

- 不能保证求得的最后解是最佳的

- 不能用来求最大值或最小值的问题
- 只能求满足某些约束条件的可行解的范围

所以贪心策略适用的前提是：局部最优策略能导致产生全局最优解。

11.2 贪心算法的应用

经典适用的场景，有 Huffman 哈夫曼编码，另外图算法中的最小生成树（Prim、Kruscal）、单元最短路径（狄克斯特拉 Dijkstra）都用到了贪心策略。

11.2.1 哈夫曼树

在二叉树中，不同的深度的节点，在查询访问时耗费的时间不同。怎样让二叉树的查询效率更高呢？一个简单的想法是，如果能统计出节点被访问的频率，我们就可以让高频访问的节点深度小一点，以便更快速地访问到。

所以，我们可以给树中的每个节点赋予一个“权重”，代表该节点的访问频率；而从根节点到这个节点的访问路径，可以计算出它的长度，这就是访问一次的代价。把所有节点按照频率，求一个加权和，这就是我们平均一次访问代价的期望。

（1）基本概念

哈夫曼树又叫**最优二叉树**，它是由 n 个带权叶子节点构成的所有二叉树中，**带权路径长度（WPL）最短**的二叉树。

带权路径长度即为权值与路径乘积的累加，所以哈夫曼树首先是一棵二叉树；其次，构建二叉树时通过调整节点位置，使得带权路径长度最小。

下面给出哈夫曼树中的一些基本概念定义：

- 路径：指从一个节点到另一个节点之间的分支序列。
- 路径长度：指从一个节点到另一个节点所经过的分支数目。
- 节点的权：给树的每个节点赋予一个具有某种实际意义的实数，我们称该实数为这个节点的权。
- 带权路径长度：在树形结构中，我们把从树根到某一节点的路径长度与该节点的权的乘积，叫做该节点的带权路径长度。

- 树的带权路径长度（WPL）：为树中所有叶子节点的带权路径长度之和，通常记为：

$$WPL = \sum_{i=1}^{n} w_i * l_i$$

其中 n 为叶子节点的个数，ω_i 为第 i 个叶子节点的权值，l_i 为第 i 个叶子节点的路径长度。

（2）构造哈夫曼树

具体构造哈夫曼树的算法，就用到贪心策略：每次都选取当前权值最小的两个节点，让它们合并成一棵树；之后定义它们根节点的权值为左右子树之和，再让根节点和其它节点比较，最小的两个节点合并一棵树。这样，我们每一步都选取当前最优策略，最终的效果就是得到了全局最小路径权值的二叉树。这就是贪心策略的具体应用。

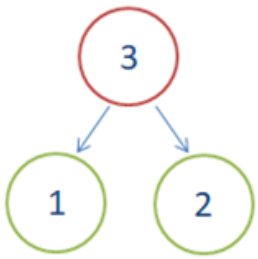
步骤如下：

（1）初始化

用给定的 n 个权值 $w_1, w_2, \ldots, w_n$ 对应的 n 个节点构成 n 棵二叉树的森林 $F = T_1, T_2, \ldots, T_n$，其中每一棵二叉树 $T_i (1 \le i \le n)$ 都只有一个权值为 w_i 的根节点，其左、右子树为空。

（2）找最小树

在森林 F 中选择两棵根节点权值最小的二叉树，作为一棵新二叉树的左、右子树，标记新二叉树的根节点权值为其左右子树的根节点权值之和。

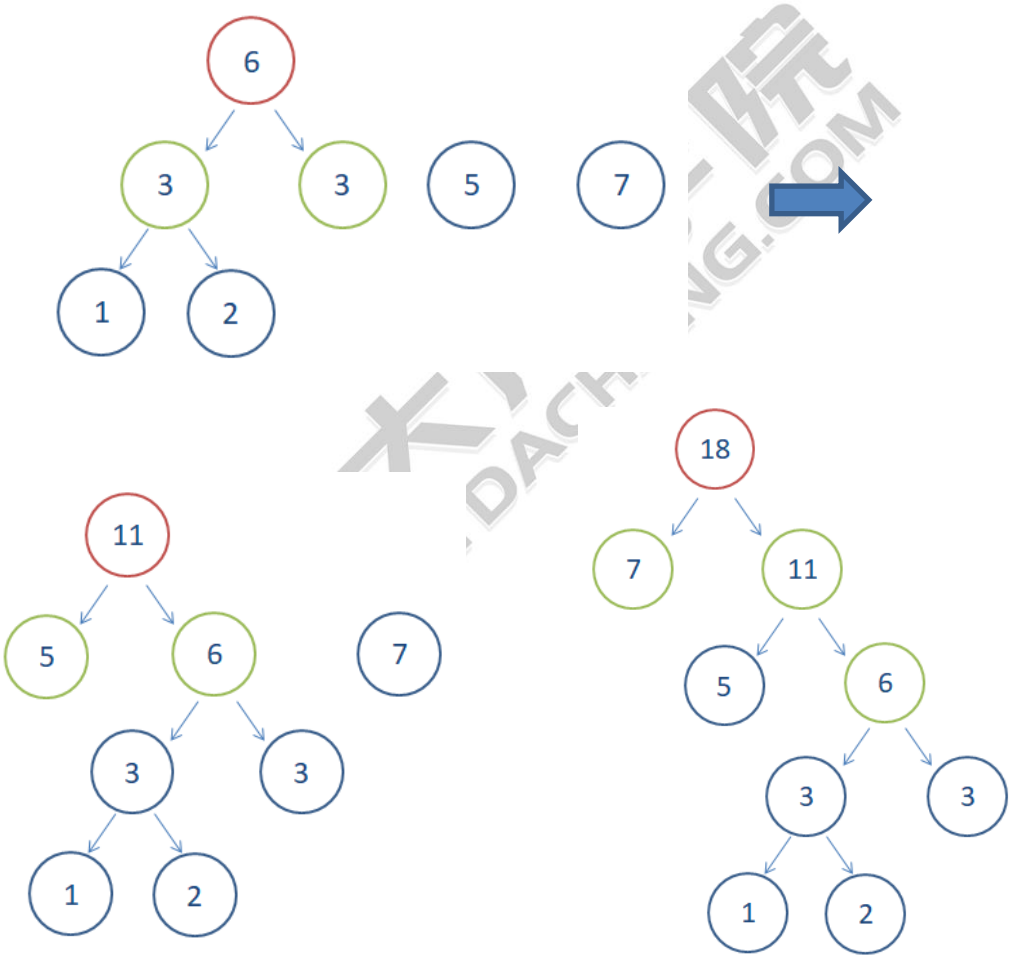


（3）删除与加入

从 F 中删除被选中的那两棵二叉树，同时把新构成的二叉树加入到森林 F 中。

（4）判断

重复（2）、（3）操作，直到森林中只含有一棵二叉树为止，此时得到的这棵二叉树就是哈夫曼树。



11.2.2 哈夫曼编码

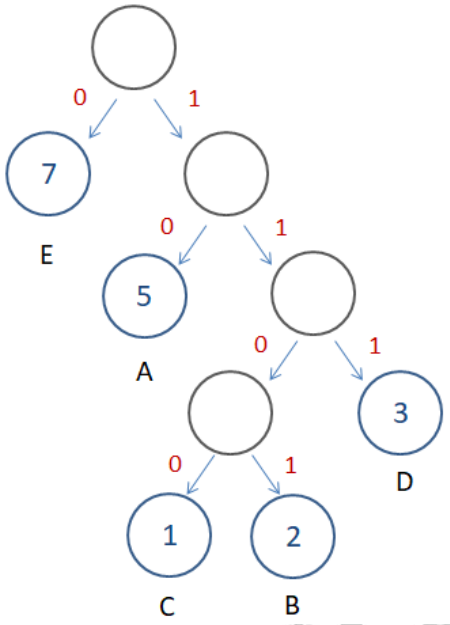
哈夫曼编码（Huffman Coding），又称霍夫曼编码，是一种编码方式，可变字长编码(VLC)的一种。这是 Huffman 于 1952 年提出一种编码方法，该方法完全依据字符出现概率来构造

异字头的平均长度最短的码字，有时称之为最佳编码，一般就叫做 Huffman 编码。

哈夫曼编码就是哈夫曼树的实际应用。主要目的，就是根据使用频率来最大化节省字符（编码）的存储空间。

比如，我们可以定义字母表[A, B, C, D, E]，每个字母出现的频率为[5, 2, 1, 3, 7]，现在希望得到它的最优化编码方式。也就是说，我们希望一段文字编码后平均码长是最短的。

构建哈夫曼树如下：



可以看到，我们的编码规则为：

A – 10　B – 1101　C – 1100　D – 111　E – 0

11.2.3 背包问题

【背包问题】有一个背包，容量是 M=150，有 7 个物品，物品可以分割成任意大小。要求尽可能让装入背包中的物品总价值最大，但不能超过总容量。

```
物品：A    B    C    D    E    F    G
重量：35   30   60   50   40   10   25
价值：10   40   30   50   35   40   30
```

分析：

目标函数：　∑pi 最大

约束条件：装入的物品总质量不超过背包容量：∑ wi <= M(M=150)

我们自然可以想到，可以考虑贪心的策略。这里有几种选择：

（1）每次挑选价值最大的物品

（2）每次挑选所占重量最小的物品

（3）每次选取**单位重量价值**最大的物品

我们的最终选择是，每次都取当前单位价值最大的物品装入背包。那么解决方案就非常简单了：选取当前剩余单位价值最大的物品放入，直到物品放完，或者背包填满。

值得注意的是，贪心算法并不是完全不可以使用，贪心策略一旦经过证明成立后，它就是一种高效的算法。

比如本题，如果题目要求物品**不能分割**，那么这就是所谓的 0-1 背包问题，就不能用贪心策略来解决了。

11.3 跳跃游戏（#55）

11.3.1 题目说明

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个位置。

示例 1:

```
输入: [2,3,1,1,4]
输出: true
```

解释: 我们可以先跳 1 步，从位置 0 到达 位置 1, 然后再从位置 1 跳 3 步到达最后一个位置。

示例 2:

```
输入: [3,2,1,0,4]
```

输出：false

解释：无论怎样，你总会到达索引为 3 的位置。但该位置的最大跳跃长度是 0，所以你永远不可能到达最后一个位置。

11.3.2 分析

这道题规定数组每个元素，就代表了可以跳跃的最大长度。

我们发现，其实只要判断出当前最远可以到达的位置，那么在这之前的元素，其实统统可以跳过。所以我们采用的，就是一个**贪心**策略：每次都判断一下当前能到达的最远位置，然后再看接下来最远可以到哪里。

11.3.3 解决方法

我们可以遍历数组中的每一个位置，并实时更新当前**最远可以到达的位置**，记为 farthest。

对于当前遍历到的位置 i，如果它小于等于当前 farthest，说明可以从起点通过若干次跳跃到达 i；接下来基于 i 最远又可以跳到 i+nums[i]，所以我们取 farthest 和 i+nums[i]的最大值，就是新的 farthest。

在遍历的过程中，如果 最远可以到达的位置 大于等于数组中的最后一个位置，那就说明最后一个位置可达，我们就可以直接返回 True 作为答案。反之，如果在遍历结束后，最后一个位置仍然不可达，我们就返回 False 作为答案。

代码如下：

```
public class JumpGame {
    // 贪心策略：维护当前能到的最远位置
    public boolean canJump(int[] nums) {
        int farthest = 0;
        // 遍历数组，更新farthest
        for (int i = 0; i < nums.length; i++ ){
            if (i <= farthest){
                farthest = Math.max(farthest, i + nums[i]);
```

```
                if (farthest >= nums.length - 1)
                    return true;
            } else {
                return false;
            }
        }
        return false;
    }
}
```

复杂度分析

时间复杂度：O(n)，其中 n 为数组的大小。只需要访问 nums 数组一遍，共 n 个位置。

空间复杂度：O(1)，不需要额外的空间开销。

11.4 跳跃游戏 II（#45）

11.4.1 题目说明

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

示例:

```
输入: [2,3,1,1,4]
输出: 2
```

解释: 跳到最后一个位置的最小跳跃数是 2。

从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。

说明:

假设你总是可以到达数组的最后一个位置。

11.4.2 分析

本题跳跃规则跟上题一致，并且已经保证可以到达最后一个位置，现在是希望得到最小的跳跃步数。

要想步数最小，很容易想到的一个思路是，每一步都迈到最大，也就是直接跳到当前能达到的最远位置。这是典型的贪心策略。

但仔细分析就会发现问题：我们每次从 i 跳到最远 farthest，相当于跳过了中间 i 到 farthest 之间的那些元素，接下来就只能按 farthest 位置的步数前进了。如果 farthest 位置的元素很小（甚至可能为 0），而中间被跳过的元素很大，那我们之前的选择明显出现了偏差。每一步的状态会影响到后续的选择，并不是“无后效”的，所以不能用这样简单的贪心策略。

11.4.3 方法一：反向跳跃

我们可以尝试用逆向思维来思考，也就是反向跳跃。

现在我们就不是从第一个位置出发了，而是从最后一个位置出发逆推。我们首先可以得到，哪些位置，可以一步直接跳到最后。

而为了让步数最少，我们可以选择让最后一次跳跃最远，也就是说，最后一跳之前所在的位置，距离最后最远。这实际上，也是一种贪心策略。

找到最后一步跳跃前所在的位置之后，我们继续贪心地寻找倒数第二步跳跃前所在的位置，以此类推，直到找到数组的开始位置。

代码如下：

```
public class JumpGameII {
    // 方法一：反向跳跃
    public int jump(int[] nums) {
        int steps = 0;
        int curPosition = nums.length - 1;
        while ( curPosition > 0 ){
            for (int i = 0; i < curPosition; i++){
```

```
                if ( i + nums[i] >= curPosition ){
                    curPosition = i;
                    steps ++;
                    break;
                }
            }
        }
        return steps;
    }
}
```

复杂度分析

时间复杂度：O(n^2)，其中 n 是数组长度。有两层嵌套循环，在最坏的情况下，数组中的所有元素都是 1，那么要寻找的“上一步”有 n 个，而每次寻找都需要遍历数组中的每个位置。

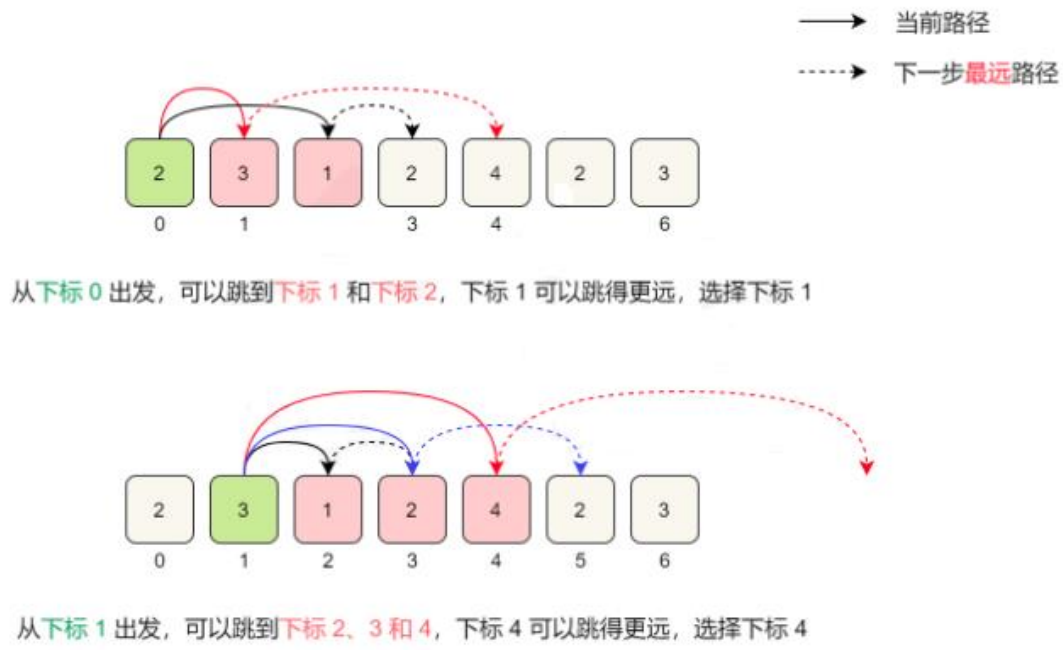
空间复杂度：O(1)。

11.4.4 方法二：正向跳跃

我们可以回忆起之前的跳跃游戏中的算法，每次都贪心地找到当前位置 i 最远能到达的位置 farthest，但不能直接跳过去，而是依次遍历接下来的每个位置，继续更新 farthest。

原因就在于，i 与 farthest 之间的元素，可能在下一步跳得很远。综合两步来看，我们并不能确定当前 i 这一步跳到最远，两步之后也跳得最远。

那自然可以想到，我们也考虑长远一点，正向推导的时候考虑两步，就可以保证最优了。这还是一个贪心策略。



那这里有一个问题，第一步选了（下标）0 -> 1，怎么保证如果选 0 -> 2 之后不会超过去呢？

我们可以看到，第二步，是肯定超不过的，因为我们第一步选下标 1，就是因为它第二步跳得远；第三步，如果基于第一步选 0 -> 2，想要跳得更远，就一定会有第二步跳到的位置 x 能跳非常远。我们可以发现，由于 x 不会超过 0 -> 1 之后最远的第二步，所以如果可以 0 -> 2 -> x，那就一定能 0 -> 1 -> x。所以第一步选位置 1 一定没有问题。

具体实现，我们可以定义双指针：一个 farthest，指向当前这一步跳跃的极限位置，它就是以当前位置为基准、跳跃一步的最远位置；另一个 nextFarthest，指向下一次跳跃的极限位置，它应该是遍历直到 farthest 的所有元素，得到基于当前位置跳跃两步的最远位置。

遍历数组，每次到达 farthest 的时候，跳跃次数加 1，并把极限位置更新为当前能到达的最远位置。

代码如下：

```
// 方法二：正向跳跃，考虑两步的最远跳跃
public int jump(int[] nums) {
```

```
    int steps = 0;
    // 定义双指针
    int farthest = 0;
    int nextFarthest = 0;
    for ( int i = 0; i < nums.length - 1; i++ ){
        nextFarthest = Math.max(nextFarthest, i + nums[i]);
        if ( i == farthest ){
            farthest = nextFarthest;
            steps ++;
        }
    }
    return steps;
}
```

复杂度分析

时间复杂度：O(n)，其中 n 是数组长度。

空间复杂度：O(1)。

11.5 任务调度器（#621）

11.5.1 题目说明

给你一个用字符数组 tasks 表示的 CPU 需要执行的任务列表。其中每个字母表示一种不同种类的任务。任务可以以任意顺序执行，并且每个任务都可以在 1 个单位时间内执行完。在任何一个单位时间，CPU 可以完成一个任务，或者处于待命状态。

然而，两个**相同种类**的任务之间必须有长度为整数 n 的冷却时间，因此至少有连续 n 个单位时间内 CPU 在执行不同的任务，或者在待命状态。

你需要计算完成所有任务所需要的**最短时间**。

示例 1：

输入：tasks = ["A","A","A","B","B","B"], n = 2

输出：8

解释：A -> B -> (待命) -> A -> B -> (待命) -> A -> B

在本示例中，两个相同类型任务之间必须间隔长度为 n = 2 的冷却时间，而执行一个任务只需要一个单位时间，所以中间出现了（待命）状态。

示例 2：

输入：tasks = ["A","A","A","B","B","B"], n = 0

输出：6

解释：在这种情况下，任何大小为 6 的排列都可以满足要求，因为 n = 0

["A","A","A","B","B","B"]

["A","B","A","B","A","B"]

["B","B","B","A","A","A"]

...

诸如此类

示例 3：

输入：tasks = ["A","A","A","A","A","A","B","C","D","E","F","G"], n = 2

输出：16

解释：一种可能的解决方案是：

A -> B -> C -> A -> D -> E -> A -> F -> G -> A -> (待命) -> (待命) -> A -> (待命) -> (待命) -> A

提示：

- 1 <= task.length <= 104
- tasks[i] 是大写英文字母
- n 的取值范围为 [0, 100]

11.5.2 分析

调度（Schedule）是操作系统的核心功能之一，保证多任务多线程的应用能够高效的利用硬件资源。本题就是模拟了操作系统中 CPU 任务调度的过程。

不同的任务可以依次执行，说明现在是单核 CPU 串行运作，同一时间只能执行一个任务。而同类型任务之间设置了冷却时间，主要也是为了保证各类型任务能够尽量均匀地占有 CPU，不要出现长时间运行同一类任务的情况。

我们发现，由于任务是串行的，所以每个任务的执行时间不可能缩短，至少需要 tasks.length 个时间单位。而现在要想总执行时间最短，其实就是让额外的冷却时间最短。

11.5.3 方法一：模拟法

一种简单的想法是，我们按照时间顺序，依次给**每一个时间单位**分配任务。这就相当于我们模拟了一个 CPU 时钟。

如果当前有多种任务可以执行，我们可以采用贪心策略：每次选择**剩余执行次数最多**的那个任务。这样将每种任务的剩余执行次数尽可能平均，就可以使得 CPU 处于待命状态的时间尽可能少。

具体实现上，我们可以定义两个状态列表，分别保存每个任务当前剩余的个数，以及最早可执行的时间。然后可以设置一个循环，模拟时间 time 不停向前推进。

具体执行过程：

- 首先遍历还没做完任务，找到可执行时间最早的那个；
- 然后判断时间 time 是否到了，如果还没到，直接推进到执行任务的时间；
- 由于有可能当前时间 time，已经超过了多个任务的最早可执行时间，所以我们还要遍历这些任务，找到可执行次数最多的那个；
- 执行任务，更新状态

代码如下：

```
public class TaskScheduler {
    // 方法一：模拟法
    public int leastInterval(char[] tasks, int n) {
        Map<Character, Integer> countMap = new HashMap<Character,
Integer>();
        for (char task : tasks) {
            countMap.put(task, countMap.getOrDefault(task, 0) + 1);
        }
        int t = countMap.size();

        // 定义两个状态List
        List<Integer> restCount = new
                        ArrayList<Integer>(countMap.values());
        List<Integer> nextAvailableTime = new
                        ArrayList<Integer>(Collections.nCopies(t, 1));
        int time = 0;
        // 遍历任务，循环执行
        for (int i = 0; i < tasks.length; i++) {
            time ++;
            // 获取所有任务中，最早可执行的时间
            int minNextAvailableTime = Integer.MAX_VALUE;
            for (int j = 0; j < t; ++j) {
                if (restCount.get(j) != 0) {
                    minNextAvailableTime = Math.min(minNextAvailableTime,
nextAvailableTime.get(j));
                }
            }
            time = Math.max(time, minNextAvailableTime);
```

```
            // 选取所有可执行任务中，剩余次数最多的那个
            int maxRestTask = -1;
            for (int j = 0; j < t; ++j) {
                if (restCount.get(j) > 0 && nextAvailableTime.get(j) <= time)
{
                    if (maxRestTask == -1 || restCount.get(j) >
restCount.get(maxRestTask)) {
                        maxRestTask = j;
                    }
                }
            }
            // 更新两个状态列表
            nextAvailableTime.set(maxRestTask, time + n + 1);
            restCount.set(maxRestTask, restCount.get(maxRestTask) - 1);
        }
        return time;
    }
}
```

复杂度分析

时间复杂度：O(N · | Σ |)，其中 N 是数组 tasks 的长度，| Σ | 是数组 tasks 中出现任务的种类。在对 time 的更新进行优化后，每一次遍历中我们都可以安排执行一个任务，所以总共进行 n 次遍历，每次遍历的时间复杂度为 O(| Σ |)，相乘即可得到总时间复杂度。

空间复杂度： O(| Σ |)。我们需要使用哈希表统计每种任务出现的次数，以及使用两个状态列表帮助我们进行遍历得到结果，这些数据结构的空间复杂度均为 O(| Σ |)。

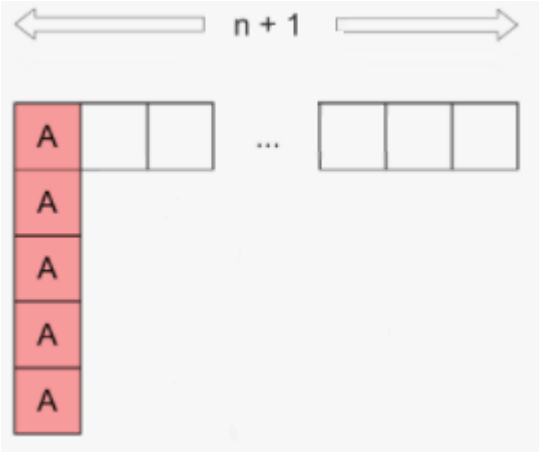
11.5.4 方法二：构造法

这种思路比较巧妙，我们可以构造一个二维矩阵，来可视化地表示每个任务执行的时间

点。

我们首先考虑所有任务种类中，执行次数最多的那一类任务，记它为 A，其执行次数为 maxCount。

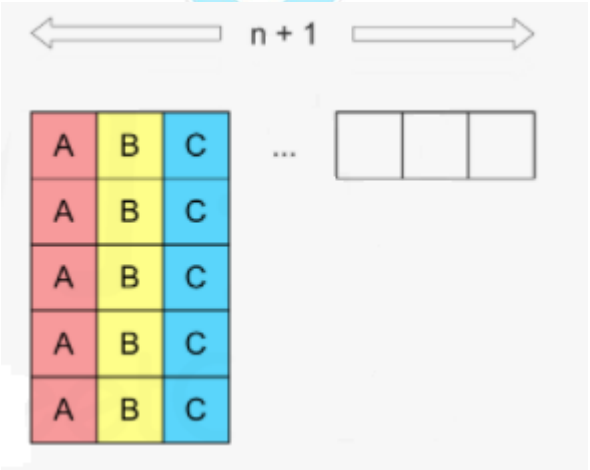
我们使用一个宽为 n+1 的矩阵，可视化地展现执行任务 A 的时间点。矩阵中任务是逐行顺序执行的，没有任务的格子对应 CPU 的待命状态。这里宽度定为 n+1 的目的，就是因为冷却时间为 n，我们只要将所有的 A 全部填入矩阵的第一列，就可以保证满足题目要求。



而且容易看出，如果只有 A 任务，这就是可以使得总时间最小的排布方法，对应的总时间为：

$$(maxCount-1) \cdot (n+1) + 1$$

同理，如果还有其它也需要执行 maxCount 次的任务，我们也需要将它们依次排布成列。例如，当还有任务 B 和 C 时，我们需要将它们排布在矩阵的第二、三列。

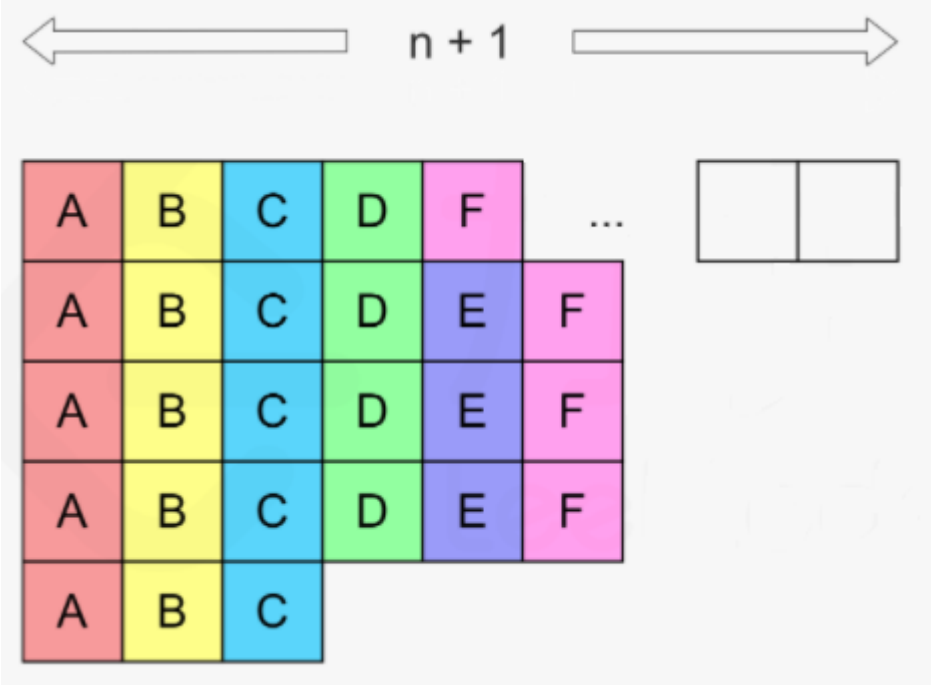


如果需要执行 maxCount 次的任务的总数量为 maxNum，那么类似地可以得到对应的总时间为：

(maxCount −1) • (n+1) + maxNum

处理完执行次数为 maxCount 次的任务，剩余任务的执行次数一定都小于 maxCount，那么我们应当如何将它们放入矩阵中呢？

一种构造的方法是，我们从**倒数第二行**开始，按照**反向列优先**的顺序（即先放入靠左侧的列，同一列中先放入下方的行），依次放入每一种任务，并且同一种任务需要连续地填入。例如还有任务 D，E 和 F 时，我们会按照下图的方式依次放入这些任务。



对于任意一种任务而言，一定不会被放入同一行两次，间隔时间一定达到了 n。因此如果我们按照这样的方法填入所有的任务，那么就可以保证总时间不变，仍然为：

(maxCount −1) • (n+1) + maxNum

代码如下：

```
// 方法二：构造法
public int leastInterval(char[] tasks, int n) {
    Map<Character, Integer> countMap = new HashMap<Character, Integer>();
    for (char task : tasks) {
        countMap.put(task, countMap.getOrDefault(task, 0) + 1);
```

```
    }
    int t = countMap.size();
    int maxCount = 0;
    int maxNum = 0;
    // 计算 maxCount
    for ( int count: countMap.values() ){
        maxCount = Math.max( maxCount, count );
    }
    // 计算 maxNum
    for ( char task: countMap.keySet() ){
        if ( countMap.get(task) == maxCount )
            maxNum ++;
    }
    return Math.max(tasks.length, (maxCount - 1) * (n + 1) + maxNum);
}
```

复杂度分析

时间复杂度：O(N +｜Σ｜)，其中 N 是数组 tasks 的长度，｜Σ｜是数组 tasks 中出现任务的种类，在本题中任务都用大写字母表示，因此｜Σ｜不会超过 26。

空间复杂度：O(｜Σ｜)。

第十二章 动态规划讲解

动态规划（Dynamic Programming，DP）是运筹学的一个分支，是求解决策过程最优化的过程。

20 世纪 50 年代初，美国数学家贝尔曼（R.Bellman）等人在研究多阶段决策过程的优化问题时，提出了著名的最优化原理，从而创立了动态规划。

动态规划的应用极其广泛，包括工程技术、经济、工业生产、军事以及自动化控制等领域，并在背包问题、生产经营问题、资金管理问题、资源分配问题、最短路径问题和复杂系统可靠性问题等中取得了显著的效果。

12.1 动态规划概念和思想

动态规划过程是：把原问题划分成多个“阶段”，依次来做“决策”，得到当前的局部解；每次决策，会依赖于当前“状态”，而且会随即引起状态的转移。

这样，一个决策序列就是在变化的状态中，“动态”产生出来的，这种多阶段的、最优化决策，解决问题的过程就称为**动态规划**（Dynamic Programming，DP）。

12.1.1 基本思想

基本思想与分治法类似，也是将待求解的问题分解为若干个子问题（阶段），按顺序求解子阶段，前一个子问题的解，会为后一子问题的求解，提供有用的信息。

在求解一个子问题的时候，我们会列出、所有可能的局部解，通过“决策”，保留那些有可能达到最优的局部解，丢弃其他局部解。依次解决每个子问题，最后一个子问题，也就是最后一个阶段的问题，那就是初始问题的解。

由于动态规划解决的问题多数有重叠子问题这个特点，为减少重复计算，对每一个子问题只解一次，将其不同阶段的不同状态保存在一个二维数组中。

与分治法最大的差别是：适合于用动态规划法求解的问题，经分解后得到的子问题往往不是互相独立的。

动态规划通常用来求解**最优化问题**。

这类问题可以有很多可行解，每个解都对应一个值，我们希望找到具有最优值（最大或

最小）的解。

12.1.2 基本步骤

动态规划所处理的问题是一个多阶段决策问题，一般由初始状态开始，通过对中间阶段决策的选择，达到结束状态。这些决策就形成了一个决策序列，同时也就确定了，完成整个过程的一条活动路线。

```
初始状态→ | 决策 1 | → | 决策 2 | →…→ | 决策 n | →结束状态
```

动态规划的设计都有着一定的模式，一般要经历以下几个步骤。

（1）划分阶段：按照问题的时间或空间特征，把问题分为若干个阶段。

（2）确定状态和状态变量：将问题发展到各个阶段时所处于的各种客观情况用不同的状态表示出来。

（3）确定决策，并写出状态转移方程

因为决策和状态转移有着天然的联系，状态转移就是：根据上一阶段的状态和决策，来导出本阶段的状态。

（4）寻找边界条件

给出的状态转移方程一般是一个递推式，需要一个递推的终止条件或边界条件。

一般，只要解决问题的阶段、状态和状态转移决策确定了，就可以写出状态转移方程。

实际应用中可以按以下几个简化的步骤进行设计：

（1）分析最优解的性质，并刻画其结构特征

（2）递归的定义最优解

（3）以自底向上或自顶向下的记忆化方式（备忘录法）计算出最优值

（4）根据计算最优值时得到的信息，构造问题的最优解

在具体应用中，最优二叉搜索树、最长公共子序列、背包问题等等都是动态规划的典型应用场景，其它一些不容易直接看出思路的题目，也往往可以用动态规划来解决。所以可以说是面试中的“杀手锏，也是一个考察重点。

12.2 从斐波那契数列说起

斐波那契数列（Fibonacci sequence），又称黄金分割数列、因数学家莱昂纳多·斐波那契（Leonardoda Fibonacci）以兔子繁殖为例子而引入，故又称为“兔子数列”，指的是这样一个数列：

```
1, 1, 2, 3, 5, 8, 13, 21, 34, …
```

它的规律是：当前数字，是之前两个数字之和。

在数学上，斐波那契数列以如下被以递推的方法定义：

```
F(0)=0，F(1)=1, F(n) = F(n - 1) + F(n - 2)（n ≥ 2，n ∈ N*）
```

12.2.1 递归实现

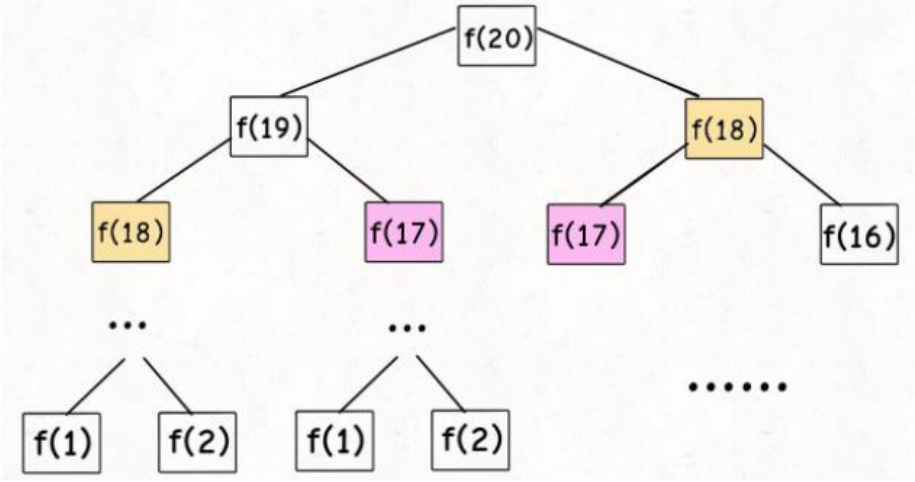
如果用代码来实现，显然用递归是顺利成章的事情：

```java
public class Fibonacci {
    // 方法一：递归
    public int fib( int n ){
        if ( n == 1 || n == 2 ) return 1;
        return fib(n-2) + fib(n-1);
    }
}
```

12.2.2 复杂度分析

可以看出，这样递归实现代码非常简洁易懂，但其实算法复杂度很高，非常低效。

这里我们假设 n = 20，可以画出一个递归树：



这是一棵二叉树，可以把原问题分成两个子问题。

也就是说，想要计算原问题 f(20)，就得先计算出子问题 f(19) 和 f(18)，然后要计算 f(19)，就要先算出子问题 f(18) 和 f(17)……以此类推。最后遇到 f(1) 或者 f(2) 的时候，结果已知，就能直接返回结果，递归树不再向下生长了。

我们发现，递归树中每个节点都代表了一次加法计算。所以当前递归子问题的个数，其实就是整个递归树节点的个数。根据二叉树的知识，节点个数为 O(2^n)。

综上，算法的时间复杂度为 O(2^n)。这是一个指数时间复杂度，运行时间随 n 增长会爆炸式上升。

考虑到递归栈的调用，空间复杂度为 O(n)。

12.2.3 动态规划实现

观察递归树，很明显可以发现算法低效的原因：存在大量重复计算。这就是可以应用动态规划问题的重要性质：具有**重叠子问题**。

我们可以做一个“备忘录”，每次算出某个子问题的答案就记到“备忘录”里；而每次遇到一个子问题时，先去“备忘录”里查找，如果发现之前已经解决过这个问题了，直接把答案拿出来用，不要再耗时去计算了。一般可以使用一个数组充当这个“备忘录”，当然也可以使用哈希表。

所以我们选取从 1 到 n 中每一步计算结果 fib(i)作为状态，保存到一个 dp 数组中。

状态转移方程，就是：

$$f(n) = \begin{cases} 1, n = 1, 2 \\ f(n-1) + f(n-2), n > 2 \end{cases}$$

当计算到 n 的时候，就可以终止了。

代码如下：

```
// 方法二：动态规划
public int fib(int n){
    if ( n == 1 || n == 2 ) return 1;
    // 定义一个 dp 数组，用来保存 fib(i) 的计算结果
    int[] dp = new int[n];
    dp[0] = dp[1] = 1;
    // 从 3 开始，依次计算 fib(i)
    for ( int i = 2; i < n; i++ ){
        dp[i] = dp[i-2] + dp[i-1];
    }
    return dp[n-1];
}
```

显然，这个过程中，我们只用到了一个循环遍历，时间复杂度为 O(n)，比起直接暴力递归的解法，有质的飞跃。

空间复杂度为 O(n)，因为用到了额外的 dp 数组来作为“备忘录”。

通过这个例子我们可以看出，所谓动态规划，其实就是对复杂问题进行划分、并对重复子问题进行一遍求解和保存、最终得到最优解的过程。

12.2.4 空间优化

我们可以发现，这里其实不需要保存数组。因为状态转移方程中，当前状态只跟之前的两个值有关，所以保存两个值就够了。

代码如下：

```
// 空间优化
public int fib(int n){
    if ( n == 1 || n == 2 ) return 1;
    // 定义两个变量，分别保存之前两个状态
    int pre2 = 1, pre1 = 1;
    for ( int i = 2; i < n; i++ ){
        int curr = pre2 + pre1;
        pre2 = pre1;
        pre1 = curr;
    }
    return pre1;
}
```

复杂度分析

时间复杂度：O(n)。

空间复杂度：O(1)，只用到了常数空间的变量。

12.3 0-1 背包问题

【0-1 背包问题】有一个背包，容量是 W=150，有 7 个物品，物品不可分割。要求尽可能让装入背包中的物品总价值最大，但不能超过总容量。

```
物品：  A   B   C   D   E   F   G
重量：35  30  60  50  40  10  25
价值：10  40  30  50  35  40  30
```

一般化表达：

一共有 N 件物品，第 i（i 从 1 开始）件物品的重量为 w[i]，价值为 v[i]。在总重量不超过背包承载上限 W 的情况下，能够装入背包的最大价值是多少？

12.3.1 问题分析

和之前提到过的背包问题不同，现在少了“物品可以分割成任意大小”这个条件。这个条件是我们使用贪心策略的关键，所以现在不能像之前那样简单处理了。我们的解决策略就是——动态规划。

（1）划分阶段、定义状态

首先我们考虑最终解的形式，每个物品只有选取、不选取两种情况，总共有 2^N 种组合。也就是说如果用暴力法遍历解空间，应该时间复杂度为 O(2^N)。这显然是不能接受的。

现在我们试图将最终的解进行分解，转化成多个阶段。我们知道，当前问题具有最优子结构，也就是说，得到最优解之后，如果从中减去一个物品 i，那么解中剩下的物品，一定是“用除 i 外的所有 N-1 件物品，放入容量为 W-w[i]的背包中”的最优解。

所以我们的目标是背包内物品的总价值，而变量是物品和背包的限重，可定义状态 dp:

```
dp[i][j]，表示将前 i 件物品，装进限重为 j 的背包，可以获得的最大价值。
0<=i<=N, 0<=j<=W
```

（2）确定决策、写出状态转移方程

我们可以将 dp[0][0...W]初始化为 0，表示将前 0 个物品（即没有物品）装入背包，最大价值为 0。那么当 i > 0 时 dp[i][j]有两种情况：

- 不装入第 i 件物品，即 dp[i−1][j]；
- 装入第 i 件物品（前提是能装下），即 dp[i−1][j−w[i]] + v[i]。

即状态转移方程为：

```
dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]]+v[i])     // j >= w[i]
```

（3）计算出局部最优值

我们可以遍历 1 ~ N 个物品、背包容量限制为 0~W 的情况。

```
dp[i-1][j]
```

和

```
dp[i-1][j-w[i]] + v[i]
```

比较取最大值。它们中的最大值，可以保证，这就是 i 个物品、背包容量 j 范围内的最优解。

（4）构造问题的最优解

最终我们要得到 dp[N][W]，就是最终的最优解

12.3.2 代码实现

```
public class KnapsackProblem {

    public int maxValue(int capacity, int[] weights, int[] values ){

        int n = weights.length;

        int[][] dp = new int[n+1][capacity+1];

        // 遍历所有可能的物品个数和背包容量

        for ( int i = 1; i <= n; i++ ){

            for ( int j = 0; j <= capacity; j++ ){

                if ( j >= weights[i-1] ){

                    dp[i][j] = Math.max( dp[i-1][j],

                                dp[i-1][j-weights[i-1]] + values[i-1] );

                } else {

                  dp[i][j] = dp[i-1][j];

                }

            }

        }

        return dp[n][capacity];

    }

}
```

复杂度分析

时间复杂度：O(NW)，其中 N 为物品个数，W 为容量取值范围。由于 W 取值总是有限的，可以认为这是“伪多项式时间”，一般情况下要优于指数时间。

空间复杂度：O(NW)。额外用到了二维数组来保存状态。

12.3.3 空间优化

我们发现，使用二维数组保存状态，空间复杂度较高。

由状态转移方程可知，dp[i][j]的值只与 dp[i-1][0,...,j-1]有关。也就是说，在 dp 矩阵里，某个值，只跟它上一行左边的值有关。

所以我们可以考虑去掉 dp 的第一维，让它退化成一维数组。在 i 取遍 1~N 时，不停地更新 dp 对应背包容量的最大值，覆盖上一轮循环结果即可。这就是动态规划中常用的方法——**滚动数组**，主要用来对空间进行优化。

代码如下：

```
// 空间优化
public int maxValue(int capacity, int[] weights, int[] values ) {
    int n = weights.length;
    // 设置一维矩阵保存状态
    int[] dp = new int[capacity+1];
    for ( int i = 1; i <= n; i++ ){
        for ( int j = capacity; j > 0; j-- ){
            if ( j >= weights[i-1] ){
                dp[j] = Math.max( dp[j], dp[j-weights[i-1]] + values[i-1] );
            }
        }
    }
    return dp[capacity];
}
```

复杂度分析

时间复杂度：O(NW)，其中 N 为物品个数，W 为容量取值范围。

空间复杂度：O(W)。只用一个数组来保存状态。

12.4 买卖股票的最佳时机（#121）

12.4.1 题目说明

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

如果你最多只允许完成一笔交易（即买入和卖出一支股票一次），设计一个算法来计算你所能获取的最大利润。

注意：你不能在买入股票前卖出股票。

示例 1:

```
输入: [7,1,5,3,6,4]
输出: 5
```

解释: 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。

注意利润不能是 7-1 = 6, 因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。

示例 2:

```
输入: [7,6,4,3,1]
输出: 0
```

解释: 在这种情况下, 没有交易完成, 所以最大利润为 0。

12.4.2 分析

最简单的想法是，直接找到最大最小值。但最小值可能在最大值之后出现，不符合题目要求。

所以我们应该先确定买入点（低点），再考虑之后的卖出点（高点）。

12.4.3 方法一：暴力法

首先可以想到暴力解法：直接遍历每一个价格，作为买入点；再遍历它之后的每一个价

格，作为卖出点，取差价最大值即可。

代码如下：

```
public class BestTimeToBuyAndSellStock {
    // 方法一：暴力法
    public int maxProfit(int[] prices) {
        int maxProfit = 0;
        // 遍历所有两两组合
        for (int i = 0; i < prices.length - 1; i++){
            for (int j = i; j < prices.length; j++){
                int curProfit = prices[j] - prices[i];
                maxProfit = Math.max(maxProfit, curProfit);
            }
        }
        return maxProfit;
    }
}
```

复杂度分析

时间复杂度：O(n^2)。双重 for 循环，耗费平方时间复杂度。

空间复杂度：O(1)。只用了常数个临时变量，没有额外的内存占用。

12.4.4 方法二：动态规划

我们发现，只要在卖出时才能真正确定利润；而要想获取最大利润，就一定要在之前的“历史最低点”买入。

所以我们可以以每一天为阶段，确定到目前为止的最优策略，并不断更新。很明显，这就是动态规划的思路。

具体策略是：遍历数组，以当天价格卖出，如果大于之前的最大利润，则更新；而最优的买入点，应该是到目前为止的历史最低价格。每天的价格可能会影响到目前为止的历史最低价格。

所以我们其实只需要保存历史最低价格以及目前可能的最大利润。

代码如下：

```
public int maxProfit(int[] prices){
    int minPrice = Integer.MAX_VALUE;
    int maxProfit = 0;
    // 遍历数组，以每天的价格作为卖出点，进行比较
    for (int i = 0; i < prices.length; i++){
        minPrice = Math.min(minPrice, prices[i]);
        maxProfit = Math.max(maxProfit, prices[i] - minPrice);
    }
    return maxProfit;
}
```

复杂度分析

时间复杂度：O(n)，只有一重循环，遍历一次数组。

空间复杂度：O(1)，只用了常数个临时变量，没有额外的内存占用。

12.5 爬楼梯（#70）

12.5.1 题目说明

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？

注意：给定 n 是一个正整数。

示例 1：

输入： 2

输出： 2

解释： 有两种方法可以爬到楼顶。

1. 1 阶 + 1 阶
2. 2 阶

示例 2：

输入： 3

输出： 3

解释： 有三种方法可以爬到楼顶。

1. 1 阶 + 1 阶 + 1 阶
2. 1 阶 + 2 阶
3. 2 阶 + 1 阶

12.5.2 分析

这是一道非常经典的面试题。爬楼梯需要一步一步爬，很明显，这可以划分阶段、用动态规划的方法来解决。

对于爬 n 级台阶，考虑它的最后一步，总是有两种情况可以到达：从 n-1 级台阶处，爬 1 个台阶；或者从 n-2 级台阶处，爬 2 个台阶。

所以我们可以将到达 n 级台阶的方法保存下来，记为 f(n)。

于是有状态转移方程：

$$f(n) = f(n-1) + f(n-2)$$

12.5.3 方法一：动态规划

代码如下：

```
public class ClimbingStairs {
    // 动态规划
    public int climbStairs(int n) {
        int[] dp = new int[n+1];
        // 初始状态
        dp[0] = 1;
```

```
        dp[1] = 1;
        // 遍历所有状态，逐级计算
        for ( int i = 2; i <= n; i++ ){
            dp[i] = dp[i-2] + dp[i-1];
        }
        return dp[n];
    }
}
```

复杂度分析

时间复杂度：O(n)，循环中的计算要执行 n-1 次。

空间复杂度：O(n)，用到了 dp 数组保存结果，长度为 n+1。

上述解法中，我们发现 dp[i]只依赖于 dp[i-1]和 dp[i-2]，所以可以用“滚动数组”的思想进行空间优化。没有必要保存之前所有的结果，只要保存两个值就够了。

代码如下：

```
// 动态规划空间优化
public int climbStairs(int n) {
    int pre2 = 1;
    int pre1 = 1;
    int curr;
    // 逐级计算
    for ( int i = 1; i < n; i++ ){
        curr = pre2 + pre1;
        pre2 = pre1;
        pre1 = curr;
    }
    return pre1;
}
```

复杂度分析

时间复杂度：O(n)，循环中的计算要执行 n-1 次。

空间复杂度：O(1)，这里只用了常数个临时变量。

12.5.4 方法二：数学方法

我们发现，状态转移方程

$$f(n) = f(n-1) + f(n-2)$$

其实就是斐波那契数列的递推公式，所以这道题目的求解，其实就是要求斐波那契数列的某一项。

通过数学方法，可以推出斐波那契数列的通项公式为：

$$f(n) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

所以，我们可以通过公式直接得到结果。

这里需要注意的是，爬楼梯的数列其实是：1，2，3，5，…

对比真正的斐波那契数列，少了第一项 1。所以对于爬 n 级台阶，结果其实是斐波那契数列的第 n+1 项。

代码如下：

```
// 方法二：数学法
public int climbStairs(int n) {
    double sqrt_5 = Math.sqrt(5);
    double fib = ( Math.pow((1 + sqrt_5) / 2, n + 1) - Math.pow((1 - sqrt_5) / 2, n + 1) ) / sqrt_5;
    return (int) fib;
}
```

复杂度分析

时间复杂度：O(1)。主要依赖于 pow 运算的时间复杂度，这依赖于 CPU 的指令集。我

们一般可以认为时间复杂度为 O(1)。

空间复杂度：O(1)。这也依赖于 pow 的实现，一般认为是 O(1)。

12.6 最长公共子序列（#1143）

12.6.1 题目说明

给定两个字符串 text1 和 text2，返回这两个字符串的最长公共子序列的长度。

一个字符串的 子序列 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。

例如，"ace" 是 "abcde" 的子序列，但 "aec" 不是 "abcde" 的子序列。两个字符串的「公共子序列」是这两个字符串所共同拥有的子序列。

若这两个字符串没有公共子序列，则返回 0。

示例 1:

```
输入：text1 = "abcde", text2 = "ace"
输出：3
```

解释：最长公共子序列是 "ace"，它的长度为 3。

示例 2:

```
输入：text1 = "abc", text2 = "abc"
输出：3
```

解释：最长公共子序列是 "abc"，它的长度为 3。

示例 3:

```
输入：text1 = "abc", text2 = "def"
输出：0
```

解释：两个字符串没有公共子序列，返回 0。

提示:

- 1 <= text1.length <= 1000

- 1 <= text2.length <= 1000
- 输入的字符串只含有小写英文字符。

12.6.2 分析

这也是一道动态规划的经典题目，很多较难的字符串问题都可以用类似的解法解决。

在生物学应用中，经常会用到比较两个 DNA 基因序列的问题。DNA 由 A、C、G、T 四种碱基分子构成，所以可以看作字母 A、C、G、T 构成的一个字符串。衡量两个 DNA 相似度的一个方法，就是看两者有多少相同的碱基对，而且，这些碱基对必须以相同的顺序出现。

抽象之后，这就变成了字符串最长公共子序列的问题。

我们发现，这个问题是有最优子结构的。

我们假设得到了字符串 str1 和 str2 的一个最长子序列 lcs。现在考虑去掉 str1 和 str2 各自的最后一个字符（记为 c1 和 c2），那么就有两种情况：

- c1 和 c2 相同，那么这也一定是 lcs 的最后一个字符（记为 t）。而且我们可以确定，去掉这个字符之后，lcs 同样是 str1 和 str2 的最长子序列。
- c1 和 c2 不同，那么 lcs 的最后一个字符 t，至少跟 c1、c2 中的一个不同。str1、str2 中与 t 不相同的那个末尾字符删除掉，不会影响结果，lcs 仍是 str1、str2 的最长子序列。

如果我们按照字符串长度来划分阶段，很明显不同阶段中的子序列，是会重复遍历的。所以，这是一个典型的具有最优子结构、而且子问题有重叠的问题，可以用动态规划来解决。

12.6.3 动态规划实现

（1）划分阶段，定义状态

分别考虑字符串 str1 和 str2 长度从 0 增长到 str.length 的过程。每一对 str1 和 str2，我们都可以求出一个当前的最长子序列（lcs）。而字符串增长之后的 lcs，会跟之前的结果相关。

为了将子问题的解保存下来，我们定义一个二维矩阵 dp，保存 str1 和 str2 在不同长度下得到的最长子序列长度。

		0	1	2	3	4	5	6
	str1 \ str2	“	b	a	b	c	d	e
0	“	0	0	0	0	0	0	0
1	a	0	0	1	1	1	1	1
2	c	0	0	1	1	2	2	2
3	e	0	0	1	1	2	2	3

这里 dp[i][j] 的含义是：对于 str1[1...i] 和 str2[1...j]，它们的 lcs 长度是 dp[i][j]。这里 str1[1...i]表示字符串 str1 截取前 i 个字符的子串，对应索引应该取 0 ~ i-1。

例如上图中，d[2][4] 的含义就是：对于 "ac" 和 "babc"，它们的 lcs 长度是 2。我们最终想得到的答案应该是 dp[3][6]。

（2）确定决策，写出状态转移方程

如果已经得到了上一阶段的最长子序列，那么接下来就是要扩充 str1 和 str2 的长度，然后比对新增字符。对于 dp[i, j]的计算，当前末尾字符为 str1[i-1]和 str2[j-1]，我们把这两个字符分别记作 c1 和 c2。可能有以下两种情况：

- c1 和 c2 相同

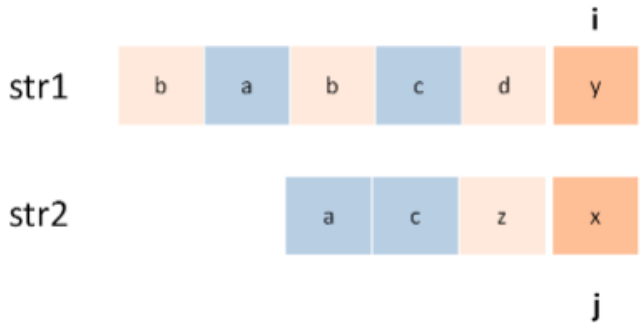
那么没有它们时，lcs 为 dp[i-1, j-1]，现在可以在后面追加一个字符了，所以

```
dp[i, j] = dp[i-1, j-1] + 1
```

- c1 和 c2 不同

那么只追加一个 c1 或者一个 c2 时，可能 lcs 会有变化，而两个都加上的时候，就不会再变化了。所以

```
dp[i, j] = max(dp[i-1, j], dp[i, j-1] )
```



（3）构造最优解

最终我们可以得到 dp 数组的右下角元素，就是全局的最优解。

代码如下：

```
public class LCS {
    // 动态规划实现
    public int longestCommonSubsequence(String text1, String text2) {
        int length1 = text1.length();
        int length2 = text2.length();
        int[][] dp = new int[length1+1][length2+1];
        // 遍历所有状态位置
        for (int i = 1; i <= length1; i++){
            for (int j = 1; j <= length2; j++){
                // 状态转移
                if ( text1.charAt(i-1) == text2.charAt(j-1) ){
                    dp[i][j] = dp[i-1][j-1] + 1;
                } else {
                    dp[i][j] = Math.max( dp[i-1][j], dp[i][j-1] );
                }
            }
        }
        return dp[length1][length2];
```

```
    }
}
```

复杂度分析

时间复杂度：O(n1*n2)。其中 n1、n2 分别为字符串 str1、str2 的长度。

空间复杂度：O(n1*n2)。用到了二维数组来保存状态。

12.7 打家劫舍（#198）

12.7.1 题目说明

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你 不触动警报装置的情况下 ，一夜之内能够偷窃到的最高金额。

示例 1：

```
输入：[1,2,3,1]
输出：4
```

解释：

偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。

偷窃到的最高金额 = 1 + 3 = 4 。

示例 2：

```
输入：[2,7,9,3,1]
输出：12
```

解释：

偷窃 1 号房屋 (金额 = 2), 偷窃 3 号房屋 (金额 = 9)，接着偷窃 5 号房屋 (金额 = 1)

偷窃到的最高金额 = 2 + 9 + 1 = 12 。

提示：

- 0 <= nums.length <= 100
- 0 <= nums[i] <= 400

12.7.2 分析

由于不能偷窃连续的房屋，我们自然想到，隔一个偷一间显然是一个不错的选择。那是不是，直接计算所有奇数项的和，以及所有偶数项的和，取最大值就可以了呢？并没有这么简单。

例如，如果是[2，7，1，3，9]，很明显，偷 2，1，9 或者 7，3 都不是最佳选择，偷 7，9 才是。

这里的关键是，对于三个连续的房屋 2，7，1，由于跟后面的 9 都隔开了，所以我们可以选择偷 2，1，也可以直接选择偷 7。这就需要分情况讨论了。

所以我们发现，从最后往前倒推，最后一间屋 n，有偷和不偷两种选择：

- 如果偷，那么前一间屋 n-1 一定没有偷，我们考虑 n-2 之前的最优选择，加上 n 就可以了；
- 如果不偷，那么 n-1 之前的最优选择，就是当前的最优选择。

所以，这明显是一个动态规划的问题。

12.7.3 动态规划实现

我们可以将前 n 个房屋能偷到的最大金额，保存到状态数组 dp。前 i 个房屋能够偷到的最大金额，就是 dp[i]。

可以得到状态转移方程：

$$dp[i] = \max(dp[i-2] + nums[i], dp[i-1])$$

代码如下：

```java
public class HouseRobber {

    // 动态规划

    public int rob(int[] nums) {

        int n = nums.length;

        if (nums == null || n == 0) return 0;


        int[] dp = new int[n + 1];

        dp[0] = 0;

        dp[1] = nums[0];

        // 遍历状态，依次转移

        for (int i = 2; i <= n; i++){

            dp[i] = Math.max( dp[i-1], dp[i-2] + nums[i-1] );

        }

        return dp[n];

    }

}
```

复杂度分析

时间复杂度：O(n)，其中 n 是数组长度。只需要对数组遍历一次。

空间复杂度：O(n)。使用数组 dp 存储状态，长度为 n+1。

12.7.4 空间优化

上述方法使用了数组存储结果。

我们通过状态方程可以发现，每间房屋的最高总金额，只和该房屋的前两间房屋的最高总金额相关。因此只要存储之前两间房屋的最高金额就可以了。

代码如下：

```java
// 动态规划空间优化

public int rob(int[] nums) {
```

```
    int n = nums.length;
    if (nums == null || n == 0) return 0;
    int pre2 = 0;
    int pre1 = nums[0];
    // 遍历状态，依次转移
    for (int i = 1; i < n; i++){
        int curr = Math.max(pre1, pre2 + nums[i]);
        pre2 = pre1;
        pre1 = curr;
    }
    return pre1;
}
```

复杂度分析

时间复杂度：O(n)，其中 n 是数组长度。只需要对数组遍历一次。

空间复杂度：O(1)。使用滚动数组，只存储前两间房屋的最高总金额，而不需要存储整个数组的结果，因此空间复杂度是 O(1)。

12.8 零钱兑换（#322）

12.8.1 题目说明

给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

你可以认为每种硬币的数量是无限的。

示例 1：

```
输入：coins = [1, 2, 5], amount = 11
输出：3
```

解释：11 = 5 + 5 + 1

示例 2：

```
输入：coins = [2], amount = 3
输出：-1
```

示例 3：

```
输入：coins = [1], amount = 0
输出：0
```

示例 4：

```
输入：coins = [1], amount = 1
输出：1
```

示例 5：

```
输入：coins = [1], amount = 2
输出：2
```

提示：

- 1 <= coins.length <= 12
- 1 <= coins[i] <= 231 - 1
- 0 <= amount <= 104

12.8.2 分析

这道题要求硬币总面值加起来等于 amount，这是一个限制条件。而硬币个数无限供应，所以我们需要做的，就是在满足限制条件的前提下，对于不同的硬币组合，选择一个数量最少的组合。

最简单的想法，就是直接暴力列出所有可能的情况。我们可以依次考虑每种硬币，先计算出每种硬币可能的最大个数，然后遍历每种可能，选择其中面值和为 amount 的组合，并且取出数量最小的那种情形。

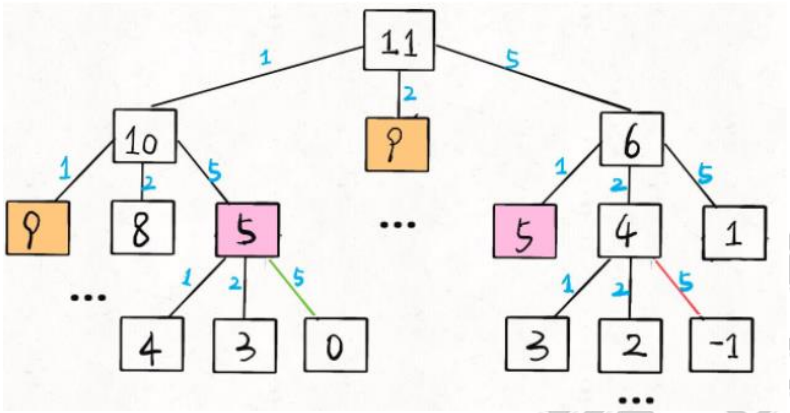
由于每种硬币可能的最大个数和 amount 有关，所以时间复杂度为 O(A^n)。其中 A 为 amount 的值，n 为硬币的种类（coins 的长度）。

暴力法的时间复杂度为指数级，这在一般场景下是不可接受的。

为了改进算法性能，我们可以考察凑出 amount 面值的具体过程：

每一步增加一个硬币，硬币个数加 1，而面值就向着目标增加对应的值。凑出 amount 时，最后一步不外乎 n 种情况，n 就是硬币的种类。

所以我们可以看出来，这就是一个典型的动态规划问题。



12.8.3 动态规划实现

我们定义一个状态数组 dp，保存总面值金额为 0~amount 的硬币最小个数。dp[i]就表示总面值为 i 的硬币最小个数。

那么状态转移方程就是：

$$dp(n) = \min_{coin \in coins} dp(n - coin) + 1$$

代码如下：

```
// 动态规划
public int coinChange(int[] coins, int amount) {
    int n = coins.length;
    int[] dp = new int[amount + 1];
    dp[0] = 0;

    for (int i = 1; i <= amount; i++){
```

```
        int minCoinNum = Integer.MAX_VALUE;
        // 遍历所有硬币面值，作为可能的“最后一步”
        for (int coin: coins){
            if (coin <= i && dp[i - coin] != -1){
                minCoinNum = Math.min(minCoinNum, dp[i - coin] + 1);
            }
        }
        dp[i] = minCoinNum == Integer.MAX_VALUE ? -1 : minCoinNum;
    }
    return dp[amount];
}
```

复杂度分析

时间复杂度：O(An)，其中 A 为题目所给的总金额 amount，n 是硬币的种类（coins 长度）。一共需要计算 O(A) 个状态，而对于每个状态，每次需要枚举 n 种硬币面值，所以一共需要 O(An)的时间复杂度。

空间复杂度：O(A)。使用 dp 数组保存状态，长度为总金额 amount + 1 的空间。

第十三章 回溯算法讲解

回溯算法实际上一个类似枚举的搜索尝试过程，主要是在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时，就 “回溯” 返回，尝试别的路径。

13.1 回溯的概念和思想

13.1.1 基本概念

回溯法是一种选优搜索法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法，而满足回溯条件的某个状态的点称为 “回溯点”。许多复杂的，规模较大的问题都可以使用回溯法，有“通用解题方法”的美称。

13.1.2 基本思想

回溯法的基本思想，其实就是不停地搜索寻找，而且是定了一个方向，就会不停向下深入，遇到岔路，就选一个一直走下去，直至此路不通，就返回再找另一条可能的岔路。

这其实就是所谓的“深度优先搜索”。深度优先搜索（DFS）一般用在图的搜索算法中，也可以用在树结构上。回溯法其实就是用了 DFS 的策略，去解决最优化问题。

一个最优化问题的解，往往可以用一个“解空间树”来表示。所以回溯法可以说就是对隐式图的深度优先搜索算法。

13.1.3 一般步骤

（1）针对所给问题，确定问题的解空间；

（2）确定节点的扩展搜索规则；

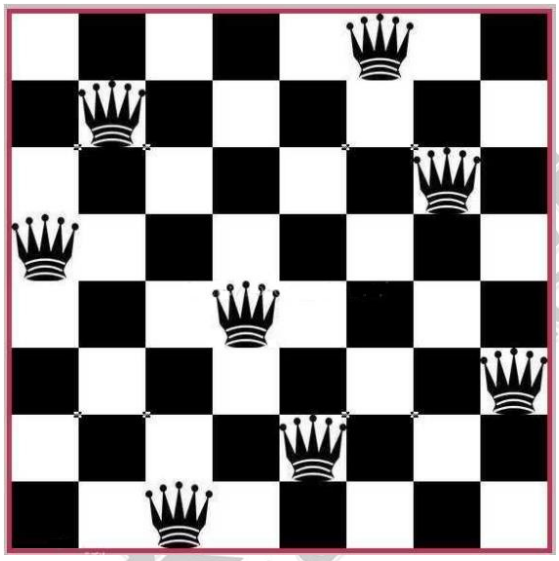
（3）以深度优先方式搜索解空间，并在搜索过程中，用剪枝函数、避免无效搜索。

我们发现，回溯可以用于所有用穷举法可以解决的问题，而动态规划只用于具有最优子结构的问题。动态规划的实质是记忆，需要储存子问题的解，回溯则不需要。所以尽管回溯

看起来类似于暴力穷举，但有些问题是无法用动态规划来解决的，就只能采用回溯的方法，比如著名的八皇后问题。

13.2 八皇后问题

在 8×8 格的国际象棋上摆放八个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上。问有多少种摆法。



八皇后问题，是一个古老而著名的问题，是回溯算法的典型案例。该问题是国际西洋棋棋手马克斯·贝瑟尔于 1848 年提出。 高斯认为有 76 种方案。1854 年在柏林的象棋杂志上不同的作者发表了 40 种不同的解，后来有人用图论的方法解出 92 种结果。计算机发明后，有多种计算机语言可以解决此问题。

13.2.1 问题分析

设八个皇后为 x_i，分别在第 i 行(I = 1，2，3，4……，8)。

- 问题的解状态：可以用

```
(1,x1)，(2,x2)，……，(8,x8)
```

表示 8 个皇后的位置。由于行号固定，可简单记为：

```
(x1,x2,x3,x4,x5,x6,x7,x8)
```

所以我们可以直接用一个长度为 8 的数组，来表示八皇后问题的一组解。

- 问题的解空间：

$(x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8)$，$1 \leqslant x_i \leqslant 8 (i=1，2，3，4\cdots\cdots，8)$

共 8^8 个状态。

- 约束条件：八个皇后位置 (1,x1)，(2,x2)，……，(8,x8)不在同一行、同一列和同一对角线上。

13.2.2 方法一：暴力穷举

每一行放一个皇后，可以放在第 1 列，第 2 列，……，直到第 8 列。穷举所有的可能，检验皇后之间是否会相互攻击。

毫无疑问，这种方法是非常低效率的，因为它并不是哪里有冲突就调整哪里，而是盲目地按既定顺序枚举所有的可能方案。

代码如下：

```
public class EightQueens {
    // 方法一：暴力枚举
    public List<int[]> eightQueens1(){
        ArrayList<int[]> result = new ArrayList<>();
        // 用一个数组保存一组解
        int[] solution = new int[8];
        // 遍历解空间
        for (solution[0] = 0; solution[0] < 8; solution[0]++){
            for (solution[1] = 0; solution[1] < 8; solution[1]++){
                for (solution[2] = 0; solution[2] < 8; solution[2]++){
                    for (solution[3] = 0; solution[3] < 8; solution[3]++){
                        for (solution[4] = 0; solution[4] < 8;
solution[4]++){
                            for (solution[5] = 0; solution[5] < 8;
```

```
solution[5]++){
                                for (solution[6] = 0; solution[6] < 8;
solution[6]++){
                                    for (solution[7] = 0; solution[7] < 8;
solution[7]++){
                                        if (check(solution))
                                    result.add(Arrays.copyOf(solution, 8));
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
        return result;
    }

    // 定义一个判定有效的方法
    private boolean check(int[] a){
        // 任意两个皇后位置比较
        for (int i = 0; i < 7; i++){
            for (int j = i + 1; j < 8; j++){
                if (a[i] == a[j] || Math.abs(a[i] - a[j]) == j - i )
                    return false;
            }
        }
        return true;
```

```
    }
}
```

复杂度分析

时间复杂度：O(N^N)。这里 N 为皇后数量，即 n 皇后问题的维度，本题 N=8。

八皇后问题如果用穷举法，需要尝试 $8^8 = 16,777,216$ 种情况。而这里的 check 方法，又需要 $C^2_8 = 28$ 次比较。

13.2.3 方法二：回溯法

回溯算法优于穷举法。

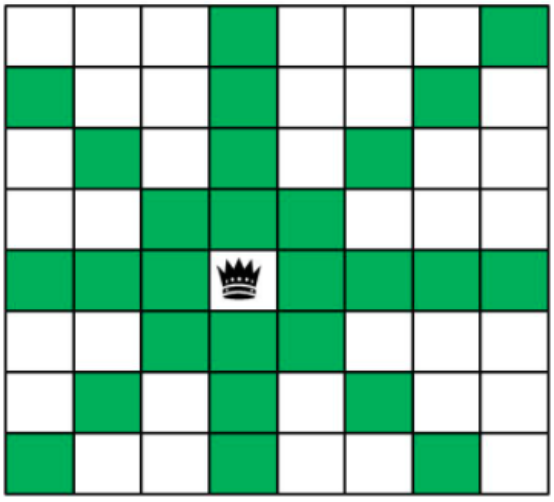
首先，将第一行的皇后放在第一列。之后第二行的皇后，也从放在第一列开始判断，这时已经发生冲突。于是调整第二行的的皇后到第二列，继续冲突就放第三列，直到不冲突为止。

如此可依次放下后续每一行的皇后。当发现某一行的皇后无处放置时，就回溯到上一行，将皇后位置向后继续调整到另一个不冲突的地方。如果上一行也无处放置，继续向上回溯。

直到每一行都无法继续放置，遍历结束。

- 判断冲突的方法改进

我们发现，一个皇后是否跟其它有冲突，主要取决于当前位置所在的横、纵、斜 4 条线。

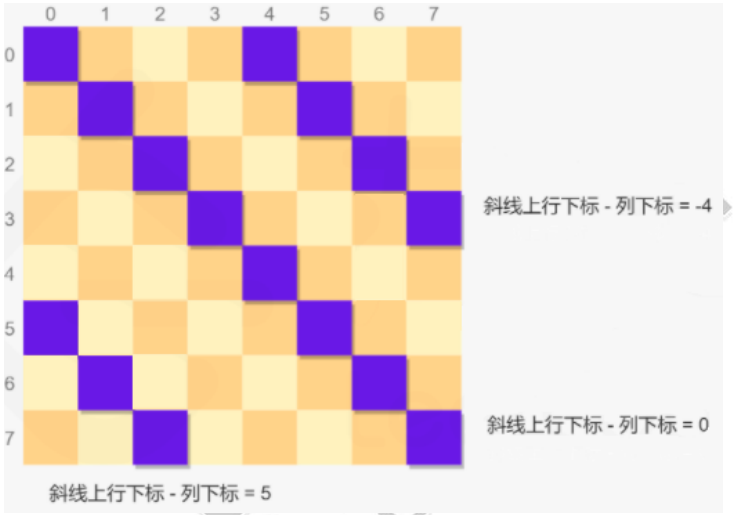


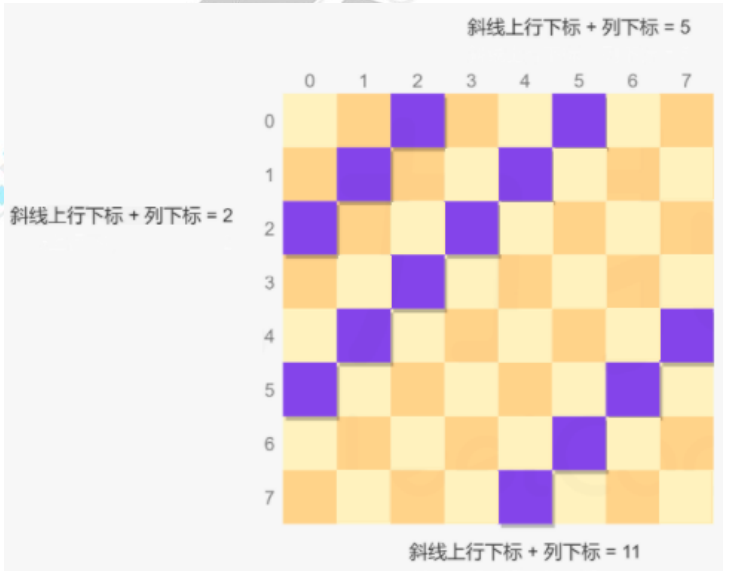
目前我们直接考虑每一行放置一个皇后，那么横向直线就不用考虑了，只需要考虑其它

三条线。

纵向比较简单，只要判断同一列是否有皇后就可以了；而对斜向仔细研究规律，可以发现，同一斜线上的格子，横纵坐标是有规律的：

- 方向一的斜线为从左上到右下方向，同一条斜线上的每个位置满足**行下标与列下标之差相等**
- 方向二的斜线为从右上到左下方向，同一条斜线上的每个位置满足**行下标与列下标之和相等**





所以为了在代码中快速判断，当前某个皇后位置是否有效，可以增加三个辅助集合：

- col：记录某一列上是否出现过皇后；
- diag1：记录某一左上-右下方向的斜线上，是否出现过皇后；
- diag2：记录某一右上-左下方向的斜线上，是否出现过皇后。

每次放置皇后时，对于每个位置判断其是否在三个集合中，如果三个集合都不包含当前位置，则当前位置是可以放置皇后的位置。

代码如下：

```
// 定义辅助集合
HashSet<Integer> cols = new HashSet<>();
HashSet<Integer> diags1 = new HashSet<>();
HashSet<Integer> diags2 = new HashSet<>();


// 方法二：回溯法
public List<int[]> eightQueens(){
    ArrayList<int[]> result = new ArrayList<>();
    int[] solution = new int[8];
    Arrays.fill(solution, -1);    // 初始填充-1
    // 传入行号0，开始调用
    backtrack(result, solution, 0);
    return result;
}
// 定义一个回溯方法
private void backtrack(ArrayList<int[]> result, int[] solution, int row){
    if (row >= 8){
        result.add(Arrays.copyOf(solution, 8));
    } else {
        // 遍历每一列，考察可能的皇后位置
        for (int column = 0; column < 8; column ++){
            if (cols.contains(column))
                continue;
            int diag1 = row - column;
```

```
            if (diags1.contains(diag1))
                continue;
            int diag2 = row + column;
            if (diags2.contains(diag2))
                continue;
            solution[row] = column;    // 当前位置可以放置皇后
            cols.add(column);
            diags1.add(diag1);
            diags2.add(diag2);
            // 递归调用，找下一行的皇后
            backtrack(result, solution, row + 1);
            // 回溯状态
            solution[row] = -1;
            cols.remove(column);
            diags1.remove(diag1);
            diags2.remove(diag2);
        }
    }
}
```

复杂度分析

时间复杂度：O(N!)，其中 N 是皇后数量。回溯的过程，其实就是 N 的一个全排列。

空间复杂度：O(N)，其中 N 是皇后数量。空间复杂度主要取决于递归调用层数、记录每行放置的皇后的列下标的数组以及三个集合，递归调用层数不会超过 N，数组的长度为 N，每个集合的元素个数都不会超过 N。

13.3 全排列（#46）

13.3.1 题目说明

给定一个 没有重复 数字的序列，返回其所有可能的全排列。

示例:

```
输入: [1,2,3]
输出:
[
  [1,2,3],
  [1,3,2],
  [2,1,3],
  [2,3,1],
  [3,1,2],
  [3,2,1]
]
```

13.3.2 分析

很明显，n 个不同的数的全排列，应该有 n！种情形。

这个问题需要暴力穷举。从前到后依次遍历每一个“位置”，每次填入一个数；而之后的一个位置，能够填入的可能性就会少一个。这样，直接用 n 重循环，考察每个位置就可以得到结果。

不过针对本题，数组的长度是不固定的，直接用多重循环的方式显然不妥。我们可以考虑使用递归。每个位置选取某个值，然后递归地对后面位置取值；由于本位置应该还可以取别的多个值，所以应该在一次递归完毕后，还可以“回到”递归调用的地方，继续取下一个值。所以，这就是回溯遍历的方法。

13.3.3 回溯实现

我们可以定义一个递归函数 backtrack(int i) ，表示当前要对第 i 个位置进行填充。

递归函数分为两个情况：

- 如果 i >= n，说明我们已经填完了 n 个位置，找到了一个可行的解，我们将 solution 放入 result 数组中，递归结束。
- 如果 i < n，我们可以选择一个尚未使用的数填入当前位置。然后递归调用，对下一个位置进行判断处理。

为了快速判断某个数“尚未使用”，可以用一个 set 来保存已经填充过的数，使用某个数时直接判断一下是否存在就可以了。

代码如下：

```
public class Permutation {
    // 定义一个辅助集合，保存当前已经用过的数
    HashSet<Integer> filledNums = new HashSet<>();
    public List<List<Integer>> permute(int[] nums){
        ArrayList<List<Integer>> result = new ArrayList<>();
        // 定义一个可行的排列
        ArrayList<Integer> solution = new ArrayList<>();
        // 从 0 位置开始填充
        backtrack(nums, result, solution, 0);
        return result;
    }
    // 定义一个回溯方法
    public void backtrack(int[] nums, List<List<Integer>> result,
List<Integer> solution, int i){
        int n = nums.length;
        if (i >= n){
```

```
            result.add(new ArrayList<>(solution));
        } else {
            // 遍历所有元素，考察下一个可选择的数
            for (int j = 0; j < n; j++){
                if (!filledNums.contains(nums[j])){
                    solution.add(nums[j]);
                    filledNums.add(nums[j]);
                    // 递归调用，继续填充后面的位置
                    backtrack(nums, result, solution, i + 1);
                    // 回溯
                    solution.remove(i);
                    filledNums.remove(nums[j]);
                }
            }
        }
    }
}
```

复杂度分析

时间复杂度：O(n*n!)，其中 n 为 nums 长度。递归调用的总次数，就是 n 个数的全排列数，而每一次调用，都需要复制 solution 到结果中，这需要耗费 O(n)的时间。所以总体的时间复杂度为 O(n!) * O(n) = O(n*n!)。

空间复杂度：O(n)，其中 n 为 nums 长度。除结果数组外，用到了辅助的 HashSet，额外的内存占用为 O(n)；另外，递归调用的深度为 O(n)，所以需要 O(n)栈空间。总共的空间复杂度就是 O(n)。

13.3.4 代码改进

上面的算法比较容易想到，不过额外使用了一个辅助的 HashSet 来判断元素是否被填充过，代码也显得比较复杂。

我们发现，solution 中的元素，其实是逐渐填入的，当前用过的数，就都在 solution 的前面填充着。所以可以在开始时，将 nums 复制到 solution 中。然后在 solution 中每填充一个数据，就把所填数和前面的元素交换。

这样，我们得到的，依然是原始 nums 那些元素。不过现在是一个被“分割”的数组，前面部分是已经填充过的；而从当前要考虑的 i 位置开始，后面是没有填充过的。

而对于回溯操作，也非常简单：只要把之前交换的两数，再换回来就可以了。

代码如下：

```
// 代码改进
public List<List<Integer>> permute(int[] nums){
    ArrayList<List<Integer>> result = new ArrayList<>();
    // 定义一个可行的排列，并复制 nums 填入
    ArrayList<Integer> solution = new ArrayList<>();
    for (int num: nums) {
        solution.add(num);
    }
    // 从 0 位置开始填充
    backtrack(result, solution, 0);
    return result;
}
public void backtrack(List<List<Integer>> result, List<Integer> solution,
int i){
    int n = solution.size();
    if (i >= n){
        result.add(new ArrayList<>(solution));
    } else {
        for (int j = i; j < n; j++){
            Collections.swap(solution, i, j);
```

```
            // 递归调用，继续填充后面的位置
            backtrack(result, solution, i + 1);
            // 回溯
            Collections.swap(solution, i, j);
        }
    }
}
```

复杂度分析

时间复杂度：O(n*n!)，其中 n 为 nums 长度。递归调用的总次数，就是 n 个数的全排列数，而每一次调用，都需要复制 solution 到结果中，这需要耗费 O(n)的时间。所以总体的时间复杂度为 O(n!) * O(n) = O(n*n!)。

空间复杂度：O(n)，其中 n 为 nums 长度。除结果数组外，递归调用的深度为 O(n)，所以需要 O(n)栈空间。总共的空间复杂度就是 O(n)。

13.4 电话号码的字母组合（#17）

13.4.1 题目说明

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。

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:

输入："23"

输出：["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

说明:

尽管上面的答案是按字典序排列的，但是你可以任意选择答案输出的顺序。

13.4.2 分析

给定数字之后，我们需要遍历每个数字对应字母的所有可能，然后进行组合。题目要求我们返回所有可能的字母组合，所以需要穷举所有解。

于是自然想到，我们可以用回溯算法来解决。依次遍历数字，选取可能的字母；递归地进行搜索，直到找到一个可行解，然后进行回溯继续遍历。

13.4.3 代码实现

可以用一个哈希表把数字对应的字母存储起来，方便快速查询。

代码如下：

```
public class LetterCombinationsOfPhoneNumber {
    // 定义一个HashMap，保存数字对应的字母
    HashMap<Character, String> numberMap = new HashMap<Character, String>()
{
        {
            put('2', "abc");
            put('3', "def");
            put('4', "ghi");
            put('5', "jkl");
            put('6', "mno");
            put('7', "pqrs");
            put('8', "tuv");
            put('9', "wxyz");
```

```
        }
    };
    public List<String> letterCombinations(String digits) {
        ArrayList<String> result = new ArrayList<>();
        if ("".equals(digits)) return result;
        StringBuffer combination = new StringBuffer();
        // 从第一个数字开始回溯处理
        backtrack(digits, result, combination, 0);
        return result;
    }
    // 定义一个回溯方法
    public void backtrack(String digits, List<String> result, StringBuffer
combination, int i){
        int n = digits.length();
        if (i >= n){
            result.add(combination.toString());
        } else {
            char digit = digits.charAt(i);
            String letters = numberMap.get(digit);
            // 遍历所有可能的字母
            for (int j = 0; j < letters.length(); j++){
                combination.append(letters.charAt(j));
                // 递归调用，继续处理后续数字
                backtrack(digits, result, combination, i + 1);
                // 回溯
                combination.deleteCharAt(i);
            }
        }
```

```
    }
}
```

复杂度分析

时间复杂度：O(3^m * 4^n)，其中 m 是输入中对应 3 个字母的数字个数，n 是输入中对应 4 个字母的数字个数，m+n 就是输入数字的总个数。需要遍历每一种字母组合，总共是 3^m * 4^n 种组合。

空间复杂度：O(m+n)。除了返回结果外，空间复杂度主要取决于回溯的递归调用深度，最大为 m+n。而哈希表的大小与输入无关，可以看成常数。

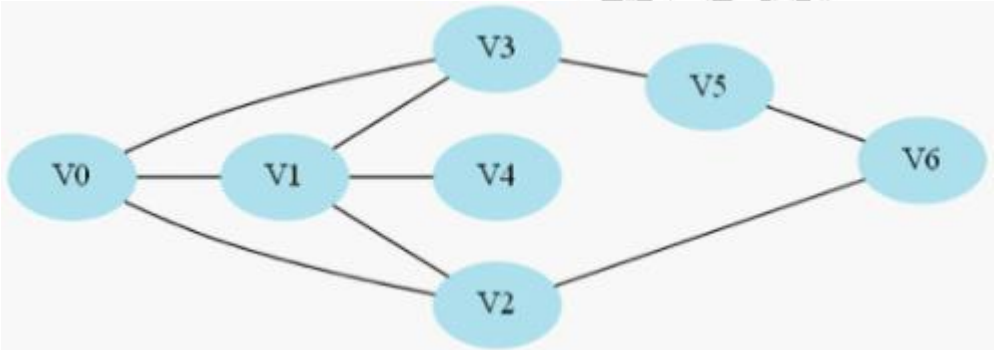
第十四章 深度优先搜索和广度优先搜索

14.1 深度优先搜索（DFS）

14.1.1 基本概念

深度优先搜索算法（Depth-First-Search， DFS）是一种用于遍历或搜索树或图的算法。沿着树的深度遍历树的节点，尽可能深的搜索树的分支。当节点 v 的所在边都已被探寻过，搜索将回溯到发现节点 v 的那条边的起始节点。

深度优先搜索是图论中的经典算法，利用深度优先搜索算法可以产生目标图的相应拓扑排序表，利用拓扑排序表可以方便的解决很多相关的图论问题，如最大路径问题等等。



DFS 可以认为是回溯法在树和图结构上的应用。当回溯用于树的时候，就是深度优先搜索。树的先序、中序、后序遍历，都属于 DFS。

14.1.2 代码实现

DFS 的代码实现，分为递归实现和非递归实现两种。

- 递归实现：访问每一个节点，然后依次递归地访问它相邻的节点。递归实现是最符合直觉的：依次处理每种可能的路径，利用递归一路探寻到底。

以树的先序遍历为例：

```
public static void preOrderRecursive( TreeNode root ){
    if (root == null) return;
```

```
    visit(root);    // 先访问根
    preOrderRecursive( root.left );    // 递归访问左子树
    preOrderRecursive( root.right );    // 递归访问右子树
}
```

- 非递归实现：需要利用循环进行深度搜索。可以利用一个栈（Stack）数据结构，在访问时弹栈，通过对其左右子节点的入栈、出栈操作，实现不同顺序的遍历访问。

同样是树的先序遍历：

```
// 先序遍历的非递归实现
public static void preOrderNonRecursive( TreeNode root ){
    // 用一个栈来辅助操作
    Stack<TreeNode> stack = new Stack<>();
    stack.push(root);
    while (!stack.isEmpty()){
        TreeNode node = stack.pop();    // 栈顶元素弹出
        visit(node);    // 先访问根
        if (node.right != null) stack.push(node.right);    // 右子树根入栈
        if (node.left != null) stack.push(node.left);    // 左子树根入栈
    }
}
```

14.1.3 复杂度分析

时间复杂度：O(n)。容易看出，不论是递归还是非递归实现，树中的节点都只会被访问一次。

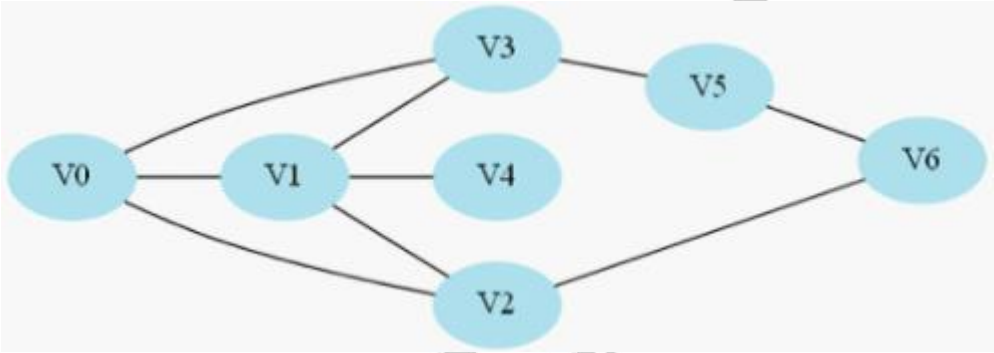
空间复杂度：O(n)。递归调用的空间占用主要是调用栈的深度，每个节点都会被递归调用一次，所以空间复杂度为 O(n)；非递归调用主要是辅助的栈空间，最坏为 O(n)。

14.2 广度优先搜索（BFS）

14.2.1 基本概念

广度优先搜索（也称宽度优先搜索，Breadth First Search，缩写 BFS）是连通图的一种遍历策略。因为它的思想是从一个顶点开始，辐射状地优先遍历其周围较广的区域，因此得名。

广度优先搜索算法是最简便的图的搜索算法之一，目的是系统地展开并检查图中的所有节点，以找寻结果。



这一算法也是很多重要的图的算法的原型。Dijkstra 单源最短路径算法和 Prim 最小生成树算法都采用了和广度优先搜索类似的思想。树的层序遍历，就是典型的 BFS。

14.2.2 代码实现

BFS 的代码实现，同样可以分为递归实现和非递归实现两种。

- 递归实现：为了方便递归，需要增加辅助方法，传入层级作为参数。

以树的层序遍历为例：

```
// 层序遍历的递归实现
public static void levelOrderRecursive( TreeNode root ){
    ArrayList<TreeNode> nodes = new ArrayList<>();
    nodes.add(root);
    bfs( nodes, 0 );
}
```

```
// 定义一个递归的辅助方法
private static void bfs(ArrayList<TreeNode> nodes, int level){
    // 基准情况
    int n = nodes.size();
    if (n == 0) return;
    ArrayList<TreeNode> nextLevelNodes = new ArrayList<>();
    // 处理当前层每一个节点
    for (TreeNode node: nodes){
        visit(node);
        if (node.left != null) nextLevelNodes.add(node.left);
        if (node.right != null) nextLevelNodes.add(node.right);
    }
    // 递归调用，处理下一层
    bfs(nextLevelNodes, level + 1);
}
```

- 非递归实现：需要利用循环进行广度搜索。可以借助队列（Queue）数据结构，每访问一个节点，就让它出队，同时让它的子节点入队。

同样是树的层序遍历：

```
public static void levelOrderNonRecursive( TreeNode root ){
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    // 只要队列不为空，就一直出队入队
    while ( !queue.isEmpty() ){
        TreeNode node = queue.poll();
        visit(node);
        if ( node.left != null )
```

```
            queue.offer(node.left);
        if ( node.right != null )
            queue.offer(node.right);
    }
}
```

很明显，对于层序遍历，非递归的实现代码要更加清晰简洁。

14.2.3 复杂度分析

时间复杂度：O(n)。不论是递归还是非递归实现，树中的节点都只会被访问一次。

空间复杂度：O(n)。递归调用栈的深度就是树的深度，为 O(logn)；但每层都需要一个额外的列表来保存节点，一层节点最多为 n/2，所以空间复杂度为 O(n)。对于非递归调用，主要是辅助的队列空间，最坏为 O(n)。

14.3 二叉树的序列化与反序列化（#297）

14.3.1 题目说明

序列化是将一个数据结构或者对象转换为连续的比特位的操作，进而可以将转换后的数据存储在一个文件或者内存中，同时也可以通过网络传输到另一个计算机环境，采取相反方式重构得到原数据。

请设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。

示例:

你可以将以下二叉树：

```
    1
   / \
```

```
    2   3
       / \
      4   5


序列化为 "[1,2,3,null,null,4,5]"
```

提示: 这与 LeetCode 目前使用的方式一致，详情请参阅 LeetCode 序列化二叉树的格式。你并非必须采取这种方式，你也可以采用其他的方法解决这个问题。

说明: 不要使用类的成员 / 全局 / 静态变量来存储状态，你的序列化和反序列化算法应该是无状态的。

14.3.2 分析

本题需要实现二叉树这种数据结构的序列化和反序列化，这类似一个编解码的过程。

序列化就是编码的过程，需要把树结构中的节点按照一定顺序遍历，然后将所有 val 以逗号连接，保存到一个字符串中；反序列化则类似解码，我们需要从字符串中，先按逗号分隔，然后解析出树中每个节点并连接成树的结构。

14.3.3 方法一：DFS

我们可以回忆树的遍历方法。对于序列化，只要用深度优先搜索的思路，先序/中序/后序遍历树中的节点，依次添加到字符串中就可以了。这里我们以先序遍历为例。

而反序列化思路也是类似，只要将过程反过来。

注意，这里所有的叶子节点，左右子节点都会保存为 null。所以对于先序遍历，开始的所有元素都是“最左侧”路径上的，直到遇到一个 null。

我们先从字符串中提取出各个节点的值，保存成列表，然后取列表头的元素作为根节点，进而递归左右子树。

代码如下：

```
public class TreeSerialization {

    // 方法一：DFS，先序遍历
```

```
// 序列化
public String serialize(TreeNode root){
    StringBuffer data = new StringBuffer();
    data.append("[");
    dfs_serialize(root, data);
    data.deleteCharAt(data.length()-1);
    data.append("]");
    return data.toString();
}

// 为了方便递归调用，单独定义一个辅助序列化方法
public void dfs_serialize(TreeNode root, StringBuffer data){
    // 基准情形
    if (root == null) {
        data.append("null,");
        return;
    }
    data.append(root.val + ",");
    // 递归处理左右子树
    dfs_serialize(root.left, data);
    dfs_serialize(root.right, data);
}

// 反序列化
public TreeNode deserialize(String data){
    // 先切分成字符串数组，然后转成 LinkedList
    String[] dataArr = data.split(",");
    LinkedList<String> dataList =
```

```
                    new LinkedList<>(Arrays.asList(dataArr));
        String firstElement = dataList.getFirst().substring(1);
        dataList.removeFirst();
        dataList.addFirst(firstElement);
        String lastElement = dataList.getLast().substring(0,
dataList.getLast().length() - 1);
        dataList.removeLast();
        dataList.addLast(lastElement);


        return dfs_deserialize(dataList);
    }


    // 定义一个反序列化辅助方法，方便递归调用
    public TreeNode dfs_deserialize(LinkedList<String> dataList){
        // 基准情形
        if (dataList.getFirst().equals("null")){
            dataList.removeFirst();
            return null;
        }
        TreeNode node = new
TreeNode(Integer.valueOf(dataList.getFirst()));
        // 删除当前节点，以当前节点为根继续递归
        dataList.removeFirst();
        node.left = dfs_deserialize(dataList);    // 后面跟着的就是左子节点
        node.right = dfs_deserialize(dataList);    // 后面的就是右子节点
        return node;
    }
}
```

复杂度分析

时间复杂度：在序列化和反序列化中，我们只访问每个节点一次，因此时间复杂度为 O(n)，其中 n 是节点数，即树的大小。

空间复杂度：在序列化和反序列化中，我们递归会使用栈空间，故渐进空间复杂度为 O(n)。

14.4.4 方法二：BFS

同样，我们也可以用广度优先的思路，对应到树的遍历上，就是层序遍历。

这里需要注意的是，序列化的时候，我们需要把每一层空余的位置填满 null，保存成完全二叉树的样子。

但是，这种方法会导致额外的空间消耗。在最坏情况下，二叉树退化为单链表，节点个数就是树的深度 n，那么对应完全二叉树节点个数为 2^n。最坏时间复杂度和空间复杂度都会达到 O(2^n)。所以实现过程略。

14.4 课程表（#207）

14.4.1 题目说明

你这个学期必须选修 numCourse 门课程，记为 0 到 numCourse-1 。

在选修某些课程之前需要一些先修课程。例如，想要学习课程 0，你需要先完成课程 1，我们用一个匹配来表示他们：[0,1]

给定课程总量以及它们的先决条件，请你判断是否可能完成所有课程的学习？

示例 1:

```
输入: 2, [[1,0]]
输出: true
```

解释: 总共有 2 门课程。学习课程 1 之前，你需要完成课程 0。所以这是可能的。

示例 2:

```
输入: 2, [[1,0],[0,1]]
```

输出：false

解释: 总共有 2 门课程。学习课程 1 之前，你需要先完成课程 0；并且学习课程 0 之前，你还应先完成课程 1。这是不可能的。

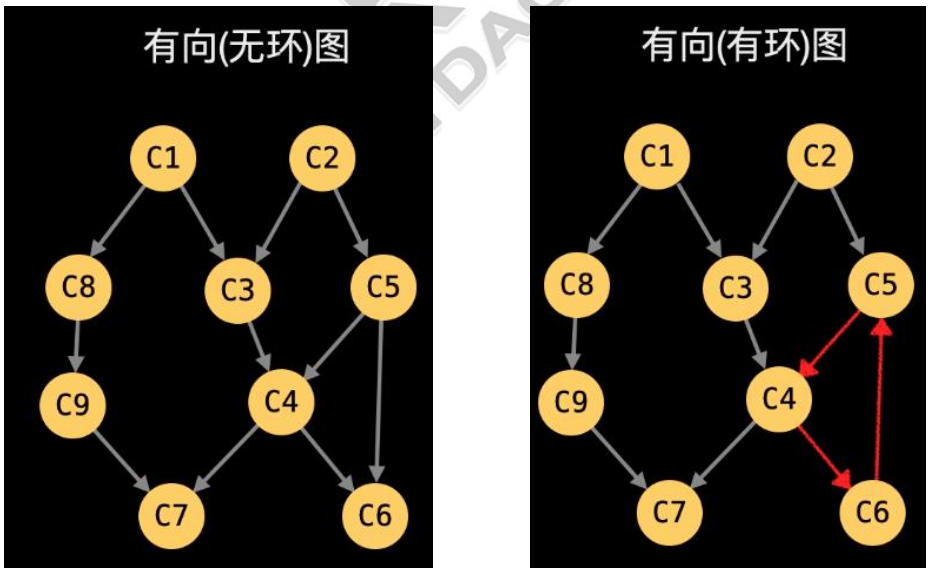
提示：

- 输入的先决条件是由 边缘列表 表示的图形，而不是 邻接矩阵 。详情请参见图的表示法。
- 你可以假定输入的先决条件中没有重复的边。
- 1 <= numCourses <= 10^5

14.4.2 分析

本题描述的是一个生活中的实际问题，也是经典的“拓扑排序”问题。

对有向无环图（DAG）G，将其中所有顶点排成一个线性序列，使得图中任意一对顶点 u 和 v，若边<u,v>∈E(G)，则 u 在线性序列中出现在 v 之前。这样的线性序列称为满足拓扑排序(Topological Order)的序列，简称拓扑序列。



对于 DAG 中的节点，一般把顶点的出边条数称为该顶点的**出度**，顶点的入边条数称为该顶点的**入度**。

我们很容易发现，如果图中有环，就肯定没有拓扑排序了。

对于本题，我们可以把给定先修课程的“匹配”[0,1]，用 DAG 上的一条边 1 -> 0 来表示，

表示“想要学习课程 0，你需要先完成课程 1”。我们的目标就是判断所有课程先修关系构成的 DAG 中是否有环。如果有环，返回 false；如果没环，应该能以每种顺序把拓扑排序列出，返回 true。

而 DAG 中是否存在环，可以通过节点的度来进行判断。

14.4.3 方法一：广度优先搜索（BFS）

最直观的想法，我们就先找出那些不需要先修课程的课，全部学完；然后继续选那些只依赖于之前修过那些课的课程。这样划分出“阶段”，依次去学习当前阶段所能学习的所有课程，完成之后再去学习下一阶段。

这其实就是广度优先搜索（BFS）的思路。而对于 BFS，最好的实现自然就是利用队列了。所以我们可以把每一阶段找到的可修课程全部入队。这些课程对应着 DAG 中入度为 0 的节点。

而为了判断某课程是否已经满足了学完所有先修课的条件，可以在每次一个课程出队时，就把它的所有后续课程入度减 1。当课程的入度减为 0 时，就代表所有先修课程已经修完，就可以将当前课程入队了。

代码如下：

```
public class CourseSchedule {


    // 方法一：BFS
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        // 定义数组保存每个节点的入度
        int[] inDegrees = new int[numCourses];
        HashMap<Integer, ArrayList<Integer>> followUpCourses = new
HashMap<>();
        // 计算入度和后续节点
        for (int[] prerequisite: prerequisites){
            inDegrees[prerequisite[0]] ++;
            ArrayList<Integer> followUpCourseList =
```

```
followUpCourses.getOrDefault(prerequisite[1], new ArrayList<>());
            followUpCourseList.add(prerequisite[0]);
            followUpCourses.put(prerequisite[1], followUpCourseList);
        }
        // 定义一个队列
        LinkedList<Integer> selectableCourses = new LinkedList<>();
        // 将当前所有入度为0的节点加入队列
        for (int i = 0; i < numCourses; i++){
            if ( inDegrees[i] == 0 ) {
                selectableCourses.offer(i);
            }
        }
        // 记录当前已学习过的课程数量
        int finishedCourseNum = 0;
        while ( !selectableCourses.isEmpty() ){
            int curCourse = selectableCourses.poll();
            finishedCourseNum ++;
            // 遍历当前课程的后续课程列表
            for ( int followUpCourse:
followUpCourses.getOrDefault(curCourse, new ArrayList<>()) ){
                inDegrees[followUpCourse] --;
                // 如果入度为0，加入队列
                if (inDegrees[followUpCourse] == 0){
                    selectableCourses.offer(followUpCourse);
                }
            }
        }
        return finishedCourseNum == numCourses;
```

```
    }
}
```

复杂度分析

时间复杂度: O(n+m)，其中 n 为课程数，m 为先修课程的条件数量。因为计算节点入度和后续课程列表需要遍历先决条件；而启动搜索扫描入度为 0 的节点需要遍历节点；每个节点最多入队、出队各一次。总计时间复杂度为 O(m+n)，这其实就是对图进行广度优先搜索的时间复杂度。

空间复杂度：O(n+m)。我们用了额外的数组来存储节点的入度，耗费空间 O(n)；用 HashMap 存储后续节点，其实就是图的“邻接表”存储形式，耗费空间为 O(n+m)。另外在广度优先搜索中，还需要额外的队列空间，最大为 O(n)。所以总空间复杂度为 O(n+m)。

14.4.4 方法二：深度优先搜索（DFS）

广度优先搜索的算法是先考虑那些入度为 0、也就是不需要任何先修课程就可以学习的课的，属于“顺向思维”；我们也可以反过来考虑出度，采取“逆向思维”。

如果希望学完所有课程，那课程一定是有“尽头”的；换句话说，总应该有一些课程，它们是学科学习的终点，没有“后续课程”。所以我们可以优先考虑找到“没有后续课程”、也就是出度为 0 的节点。由于没有课程以它们为先修课程，所以可以放到最后再学习。

这就需要沿着 DAG 中有向边的方向一直搜索到底，就是深度优先搜索（DFS）的思路。而先找到的节点，放到最后学习，符合“后进先出”的特点，可以用栈来进行保存。

在深度优先搜索的过程中，为了防止环的出现，我们还应该对“当前路径”上之前搜寻的节点有一个标记；如果搜索到了之前的节点，说明有环，直接返回 false。

代码如下：

```
// 方法二：DFS
public boolean canFinish(int numCourses, int[][] prerequisites) {
    HashMap<Integer, ArrayList<Integer>> followUpCourses = new HashMap<>();
    // 计算后续节点
```

```
    for (int[] prerequisite: prerequisites){
        ArrayList<Integer> followUpCourseList =
followUpCourses.getOrDefault(prerequisite[1], new ArrayList<>());
        followUpCourseList.add(prerequisite[0]);
        followUpCourses.put(prerequisite[1], followUpCourseList);
    }
    // 定义一个栈，保存最后学习的课程
    Stack<Integer> lastCourses = new Stack<>();
    // 定义一个数组，保存某节点是否在当前路径上出现过
    boolean[] isSearched = new boolean[numCourses];
    boolean canFinish = true;
    // 遍历每一个节点，作为起始点
    for (int i = 0; i < numCourses && canFinish; i++){
        if (!lastCourses.contains(i)){
            canFinish = dfs(followUpCourses, lastCourses, isSearched, i);
        }
    }
    return canFinish;
}
// 定义一个辅助方法，进行深度优先搜索
public boolean dfs(HashMap<Integer, ArrayList<Integer>> followUpCourses,
                   Stack<Integer> lastCourses, boolean[] isSearched, int i){
    isSearched[i] = true;
    // 遍历所有后续课程，依次递归调用
    for (int followUpCourse: followUpCourses.getOrDefault(i, new
ArrayList<>())){
        if (!isSearched[followUpCourse]){
            // 如果递归返回false，直接返回false；否则继续遍历
```

```
            if (!dfs(followUpCourses, lastCourses, isSearched,
followUpCourse))
                return false;
        } else {
            return false;    // 如果已经出现过，直接返回false
        }
    }
    // 后续课程全部搜索完毕，当前节点入栈
    lastCourses.push(i);
    // 状态回溯
    isSearched[i] = false;

    return true;
}
```

复杂度分析

时间复杂度: O(n^2+m)，其中 n 为课程数，m 为先修课程的要求数。代码中判断节点是否已经在栈内，我们用到了 contains 方法，这需要遍历栈中所有元素，所以最坏时间复杂度会达到 O(n^2+m)。

空间复杂度: O(n+m)。我们用 HashMap 存储后续节点，其实就是图的“邻接表”存储形式，耗费空间为 O(n+m)；为了方便判断当前节点是否搜索过，用到了 isSearched 数组，耗费空间 O(n)。此外在深度优先搜索时，我们还需要单独保存结果的栈以及递归调用的栈空间，这些都要耗费 O(n)的内存空间。因此总空间复杂度为 O(n+m)。

14.4.5 DFS 的优化

为了方便快速判断节点是否已经搜索完成，我们可以另外设置一个数组 isFinished，用于保存节点是否已搜索完毕进入栈内。

当然，我们可以发现这个 isFinished 数组和之前的 isSearched，其实非常类似，都表示

的是节点的一个状态。所以可以把它们合并，就用一个 int 数组 state 来表示：

- 状态默认为 0，表示“未搜索”；
- 状态 1 表示“正在搜索中”（在当前深度搜索路径中，可能有环）；
- 状态 2 表示“搜索已完成”（已入栈）

而且，由于结果只要求返回是否存在拓扑序列，我们其实没有必要单独用栈来保存结果，所以可以省去栈。

优化如下：

```
// DFS 优化
public boolean canFinish(int numCourses, int[][] prerequisites) {
    HashMap<Integer, ArrayList<Integer>> followUpCourses = new HashMap<>();
    for (int[] prerequisite: prerequisites){
        // 获取后续节点列表
        ArrayList<Integer> followUpCourseList =
followUpCourses.getOrDefault(prerequisite[1], new ArrayList<>());
        followUpCourseList.add(prerequisite[0]);
        followUpCourses.put(prerequisite[1], followUpCourseList);
    }
    // 定义一个数组，保存节点状态：0-未搜索；1-搜索中；2-已完成
    int[] state = new int[numCourses];
    boolean canFinish = true;
    // 遍历每一个节点，作为起始点
    for (int i = 0; i < numCourses && canFinish; i++){
        // 深度搜索：只搜未搜索过的
        if (state[i] == 0){
            canFinish = canFinish && dfs(followUpCourses, state, i);
        }
```

```
    }
    return canFinish;
}
// 定义一个辅助方法，进行深度优先搜索
public boolean dfs(HashMap<Integer, ArrayList<Integer>> followUpCourses,
                   int[] state, int i){
    state[i] = 1;
    // 遍历所有后续课程，依次递归调用
    for (int followUpCourse: followUpCourses.getOrDefault(i, new
ArrayList<>())){
        if (state[followUpCourse] == 0){
            if (!dfs(followUpCourses, state, followUpCourse))
                return false;
        } else if (state[followUpCourse] == 1){
            return false;
        }
    }
    // 后续课程全部搜索完毕，状态改为2
    state[i] = 2;
    return true;
}
```

复杂度分析

时间复杂度: O(n+m)，其中 n 为课程数，m 为先修课程的要求数。这其实就是对图进行深度优先搜索的时间复杂度。

空间复杂度: O(n+m)。我们用 HashMap 存储后续节点，其实就是图的“邻接表”存储形式，耗费空间为 O(n+m)；存储节点状态，需要耗费空间 O(n)。此外在深度优先搜索时，我们还需要递归调用的栈空间，需要耗费 O(n)的内存空间。因此总空间复杂度为 O(n+m)。

第十五章 位运算和数学方法讲解

15.1 二进制和位运算复习总结

15.1.1 计算机的二进制表示

计算机底层是集成数字电路，而集成电路的基础是半导体元器件。在数字电路中，每个元件只能有高电位、低电位两个稳定状态，我们把它记作 1 和 0。

所以计算机中数据的表达，就是基于很多元件的 0-1 状态的组合，这是一个以 2 为基数的计数系统，我们把它叫做**二进制**。

二进制的特点，就是只用 0 和 1 两个符号进行计数，逢 2 进位。计算机最终处理的都是二进制的数据。

以 Java 中 byte 类型为例，一个数值占据 1 个字节的内存空间，具体表现就是 8 个二进制位。例如：

$8 = (0000\ 1000)_2$　　$37 = (0010\ 0101)_2$　　$196 = (1100\ 0100)_2$

15.1.2 进制转换

我们的日常习惯，处理的都是十机制数，而计算机处理的是二进制数。为了更好地理解计算机底层的工作机制、优化算法性能，我们需要了解十进制和二进制之间的转换规则。

（1）二进制转十进制

二进制数转换成十进制数，方法是“按权展开求和”。

二进制数中的每一位，都对应的是 2 的某个整次幂。因此，我们可以直接将每一位对应的数表示为十进制，然后相加即可。

例如：

$(1000\ 1111)_2 = (2^7 + 2^3 + 2^2 + 2^1 + 2^0)_{10} = (128 + 8 + 4 + 2 + 1)_{10} = (143)_{10}$

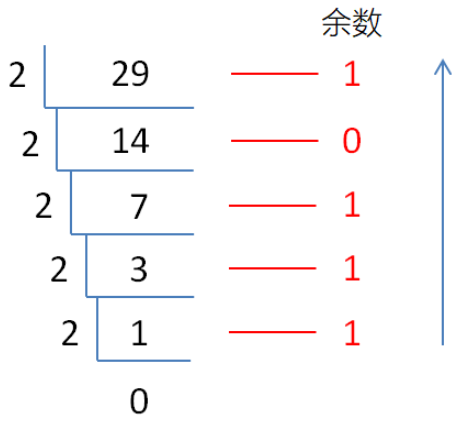
（2）十进制转二进制

十进制数转换成二进制数，方法是“除 2 取余，逆序排列”。

从二进制转十进制的过程可以看出，十进制写成 2 的整次幂展开的形式，就对应着二进制的每一位 1，所以我们可以将原数不停除以 2，得到的余数就是对应位的值。

例如：

$$(29)_{10} = (2^4 + 2^3 + 2^2 + 2^0)_{10} = (0001\ 1101)_2$$



15.1.3 原码、反码和补码

对于一个数，计算机需要保存的是它的二进制形式。但是对于有符号数，又应该怎样处理呢？原码、反码、补码就是机器存储一个有符号数的具体编码方式。

（1）原码

原码就是符号位加上数值的绝对值，即用第一位表示符号，其余位表示值。比如：如果是 8 位二进制的正 1 和负 1：

$$[+1]_{10} = [0000\ 0001]_{原}$$

$$[-1]_{10} = [1000\ 0001]_{原}$$

因为第一位是符号位，所以 8 位二进制数的取值范围就是：

$$[1111\ 1111\ ,\ \ 0111\ 1111]$$

即

$$[-127\ ,\ 127]$$

（2）反码

反码的表示方法是：

- 正数的反码是其本身；

- 负数的反码是在其原码的基础上，符号位不变，其余各个位取反。

```
[+1] = [0000 0001]原 = [0000 0001]反
[-1] = [1000 0001]原 = [1111 1110]反
```

（3）补码

补码的表示方法是：

- 正数的补码就是其本身；
- 负数的补码是在其原码的基础上，符号位不变，其余各位取反，再+1。（也就是反码加 1）

```
[+1] = [0000 0001]原 = [0000 0001]反 = [0000 0001]补
[-1] = [1000 0001]原 = [1111 1110]反 = [1111 1111]补
```

可以看得出来，不论那种编码形式，正数都是不变的；反码和补码主要用来处理负数。在计算机系统中，一般统一用补码来表示有符号数。

使用补码的好处是：

- 解决了 0 的符号问题以及 0 的两个编码问题
- 可以将符号位和数值域统一处理，加法和减法也可以统一处理
- 补码与原码相互转换，其运算过程是相同的，不需要额外的硬件电路

15.1.4 位运算符讲解

位运算就是直接对整数在内存中的二进制位进行操作。相比于普通的整数算术运算，位运算是底层运算，效率更高，代码也更加简洁。

Java 中支持的位运算有：

- &：按位与

  如果相对应位都是 1，则结果为 1，否则为 0

- |：按位或

  如果相对应位都是 0，则结果为 0，否则为 1

- ~：按位非

  按位取反运算符翻转操作数的每一位，即 0 变成 1，1 变成 0

- ^：按位异或

如果相对应位值相同，则结果为 0，否则为 1

- <<：左位移运算符

  左操作数按位左移右操作数指定的位数

- >>：右位移运算符

  左操作数按位右移右操作数指定的位数

- >>>：无符号右移运算符

  左操作数的值按右操作数指定的位数右移，移动得到的空位以零填充

15.2 2 的幂（#231）

15.2.1 题目说明

给定一个整数，编写一个函数来判断它是否是 2 的幂次方。

示例 1:

```
输入：1
输出：true
```

解释: $2^0 = 1$

示例 2:

```
输入：16
输出：true
```

解释: $2^4 = 16$

示例 3:

```
输入：218
输出：false
```

15.2.2 分析

本题是一个数学问题。在计算机系统中，2 的整次幂往往有非常重要的意义，往往是众多数据类型能够表示的数值边界，是真正的“整数”。

2 的整次幂，有很多重要的性质：

- 能被 2 连续整除
- 写成 2 进制的形式，就是一个 1 后面跟着 n 个 0（n≥0）

我们可以利用这些特性，构造出不同的解法。

15.2.3 方法一：除 2 判断余数

最简单的想法，就是借鉴十进制转换二进制的方法，不断地除以 2，判断当前余数是否为 0，只要中间出现了 1，那就不是 2 的整次幂。

这里需要注意的是，即使是 2 的整次幂，最后也应该有一次余数为 1。

那如何判断当前余数 1 是“中间产生”还是“最终产生”呢？只需要看当前的被除数是否为 1 就可以了：如果是最后一次除法，被除数应该就是 1。

代码如下：

```
public class PowerOfTwo {
    // 方法一：除 2 判断余数
    public boolean isPowerOfTwo(int n){
        if (n <= 0) return false;
        // 不停地除以 2
        while (n % 2 == 0)
            n /= 2;
        return n == 1;
    }
}
```

复杂度分析

时间复杂度：O(logn)。循环的次数，就是二进制展开的位数，也就是以 2 为底 n 的对数。

空间复杂度：O(1)。

15.2.4 方法二：位运算（与自身减 1 做位与）

对于这样一个数学问题，O(logn)的时间复杂度显然不能让我们满意。

我们可以利用之前分析的2的幂第二个特性来进行优化，也就是说：写成2进制形式后，就是一个 1 后面跟着 k 个 0（k≥0）。

这种形式有一个非常明显的性质，就是减 1 之后，就会变成 k 个 1。所以我们可以发现，对于 2 的幂，这个数自身和它减 1 之后的数，进行位与运算，得到的结果应该是 0；或者进行异或运算，得到的结果应该是 k+1 个 1。

当然很明显，位与运算的结果更好判断。写成逻辑表达式就是：

```
n & (n - 1) == 0
```

代码如下：

```
// 方法二：位运算（与自身减 1 做位与）
public boolean isPowerOfTwo(int n){
    if (n <= 0) return false;
    return (n & n - 1) == 0;
}
```

复杂度分析

时间复杂度：O(1)。只需要做一次位运算。

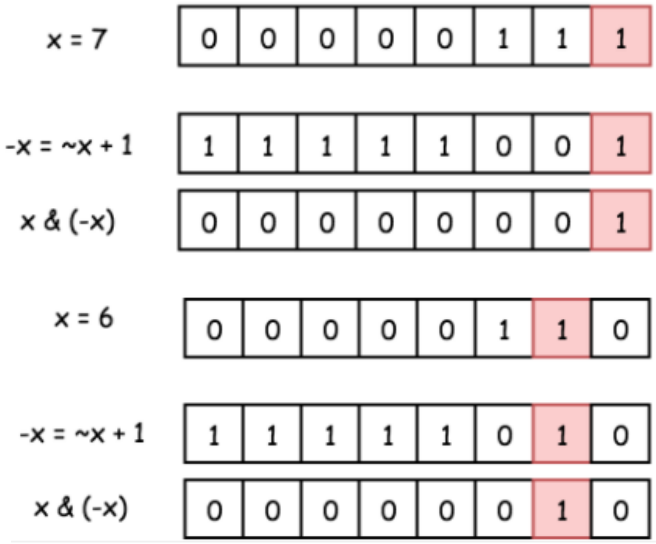
空间复杂度：O(1)。

15.2.5 方法三：位运算（与相反数做位与）

由于 2 的幂的二进制表达中，只有一个 1，所以另外一个思路是，我们可以试图获取二进制数中最右端的 1。我们现在要保留最后一位 1，其它位全部变 0，如果等于自身，那就是 2 的幂。

要想其它位都变为 0，容易想到，如果取反码，然后跟自身做位与，自然就全是 0 了。如果再加 1，就可以保留最后一位 1 了。

我们知道，在计算机底层，负数的补码表示，就是反码加 1。所以我们要做的，就是让 n 和-n 做位与运算。得到的，就只有最右面一位 1，其它位都为 0。



如果当前数 n 为 2 的幂，那么最右面的一位 1，其实就是最高位的 1；保留这一位，其实跟原数是完全相等的。

写成表达式就是：

```
n & (-n) == n
```

代码如下：

```
// 方法三：位运算（与相反数做位与）
public boolean isPowerOfTwo(int n){
    if (n <= 0) return false;
    return (n & -n) == n;
}
```

复杂度分析

时间复杂度：O(1)。

空间复杂度：O(1)。

15.3 汉明距离（#461）

15.3.1 题目说明

两个整数之间的汉明距离指的是这两个数字对应二进制位不同的位置的数目。

给出两个整数 x 和 y，计算它们之间的汉明距离。

注意：

0 ≤ x, y < 231.

示例:

```
输入: x = 1, y = 4
输出: 2
```

解释:

```
1   (0  0  0  1)
4   (0  1  0  0)
        ↑     ↑
```

上面的箭头指出了对应二进制位不同的位置。

15.3.2 分析

汉明距离（Hanming Distance）是用于统计两段信息之间差异的概念，有很多用途：在信息论中用来量化字符串的差异，在信息编码中用于错误检测。

对于两个字符串而言，汉明距离就是对应位置的不同字符的个数；对两个整数而言，汉明距离是对应位置上数字不同的位数。本题的目的，就是求出两个整数之间的汉明距离，也就是求它们二进制表示中不同的那些位的数量。

对于两个二进制数中的“某个位”，相同的不予统计，只统计那些不同的。这自然让我们联想到位运算中的“异或”：对应位相同为 0，相异为 1。

所以我们要做的，就是对两个数 x 和 y 做异或，然后统计结果中为 1 的位的个数。

15.3.3 方法一：利用语言内置方法

大多数编程语言中，都存在各种内置计算等于 1 的位数函数。例如 Java 中，Integer 类就提供了 bitCount 方法，用于统计某个整数中 1 的个数。

代码如下：

```
public class HammingDistance {
    // 方法一：直接调库
    public int hammingDistance(int x, int y) {
        return Integer.bitCount(x ^ y);
    }
}
```

复杂度分析

时间复杂度：O(1)。主要有两个操作：异或操作，花费常数时间；另外还用调用内置的 bitCount 方法。Java 中的 bitCount 只进行了有限次位移、位与和相加操作，所以时间复杂度为 O(1)。

空间复杂度：O(1)，只使用常数空间保存结果。

15.3.4 方法二：逐个移位

解决算法问题时，直接调库可能会受到面试官的质疑。所以我们最好还是自己实现一个统计位个数的方法。

最直观的想法就是，我们可以逐个右移每一位，然后判断当前最后一位是否为 1，统计个数。

判断某一位为 1 最简单的方法，就是和 1 做位与操作。当然，末位为 1，代表这是一个奇数，所以我们用算术方法取模来判断也是可以的。

代码如下：

```java
// 方法二：逐位右移
public int hammingDistance(int x, int y) {
    int xor = x ^ y;
    int count = 0;
    // 逐位右移，判断最后一位是否为1
    while ( xor != 0 ){
        // 如果跟1做位与结果是1，那么最后一位就是1
        if ( (xor & 1) == 1) count ++;
        xor >>= 1;
    }
    return count;
}
```

复杂度分析

时间复杂度：O(1)。主要考察 while 循环的次数。Java 中 int 类型是固定的 32 位，所以最多需要 32 次位移判断。时间复杂度为 O(1)。

空间复杂度：O(1)，只用到了常数个临时变量。

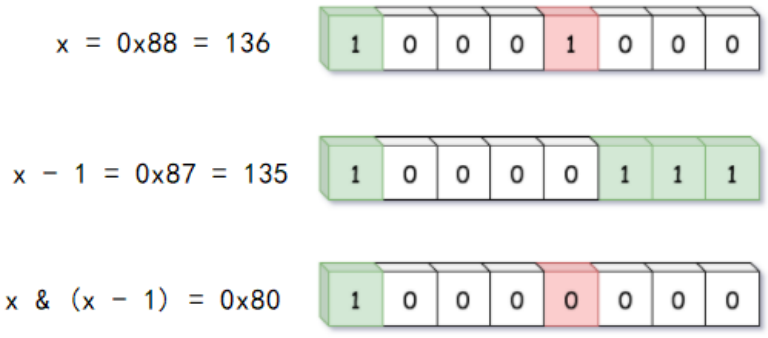
15.3.5 方法三：快速移位（布莱恩·柯尼根算法）

上面的方法我们是逐位移动，尽管时间复杂度是 O(1)，但依然显得有些繁琐。

其实我们发现，对于人类来说，用肉眼直接判断会更容易：因为我们可以直接忽略那些 0，找到一个个的 1，数出来就可以了。

这种“跳过 0”的快速计数方法，需要我们能够直接找到最右端的一个 1，然后将后面的所有 0 直接删掉。

我们可以用 x 和 x-1 进行位与，当 x 是 2 的幂时，结果就是 0；如果不是 2 的幂，其实就相当于将最后一个 1，以及后面的所有 0 消去了，前面不受影响。



这样，我们只需每次消去最右端的 1 以及它后面的 0，直到结果为 0。消去操作的次数，就是 1 的个数。

这种解法叫做**布莱恩·柯尼根位计数算法**。该算法发布在 1988 年 《C 程序设计语言第二版》（<The C Programming Language>）的练习中。

代码如下：

```
// 方法三：快速右移
public int hammingDistance(int x, int y) {
    int xor = x ^ y;
    int count = 0;
    // 快速右移，跳过最后一个 1 之后的所有 0
    while ( xor != 0 ){
        xor &= xor - 1;
        count ++;
    }
    return count;
}
```

复杂度分析

时间复杂度：O(1)。与方法二类似，主要考察 while 循环的次数。由于跳过了所有的 0，所以循环执行的次数，就是 1 的个数 count，最坏情况下为 32。相比之下，所做的迭代操作

更少了。

空间复杂度：O(1)，只用到了常数个临时变量。

15.4 可怜的小猪（#458）

15.4.1 题目说明

有 buckets 桶液体，其中 正好 有一桶含有毒药，其余装的都是水。它们从外观看起来都一样。为了弄清楚哪只水桶含有毒药，你可以喂一些猪喝，通过观察猪是否会死进行判断。不幸的是，你只有 minutesToTest 分钟时间来确定哪桶液体是有毒的。

喂猪的规则如下：

1. 选择若干活猪进行喂养
2. 可以允许小猪同时饮用任意数量的桶中的水，并且该过程不需要时间。
3. 小猪喝完水后，必须有 minutesToDie 分钟的冷却时间。在这段时间里，你只能观察，而不允许继续喂猪。
4. 过了 minutesToDie 分钟后，所有喝到毒药的猪都会死去，其他所有猪都会活下来。
5. 重复这一过程，直到时间用完。

给你桶的数目 buckets ，minutesToDie 和 minutesToTest ，返回在规定时间内判断哪个桶有毒所需的 最小 猪数。

示例 1：

```
输入：buckets = 1000, minutesToDie = 15, minutesToTest = 60
输出：5
```

示例 2：

```
输入：buckets = 4, minutesToDie = 15, minutesToTest = 15
输出：2
```

示例 3：

```
输入：buckets = 4, minutesToDie = 15, minutesToTest = 30
输出：2
```

提示：

- 1 <= buckets <= 1000
- 1 <= minutesToDie <= minutesToTest <= 100

15.4.2 分析

（1）条件提炼

这道题目中主要有四个变量：小猪的数量 n，桶的数量 buckets，冷却时间 minutesToDie 和总测试时间 minutesToTest。

容易看出，两个时间变量有密切的联系。我们把一次喂小猪喝水叫做一次“实验”，由于一次实验之后必须等待冷却时间才能得到结果，所以我们最多可以进行

```
k = minutesToTest / minutesToDie
```

次实验。

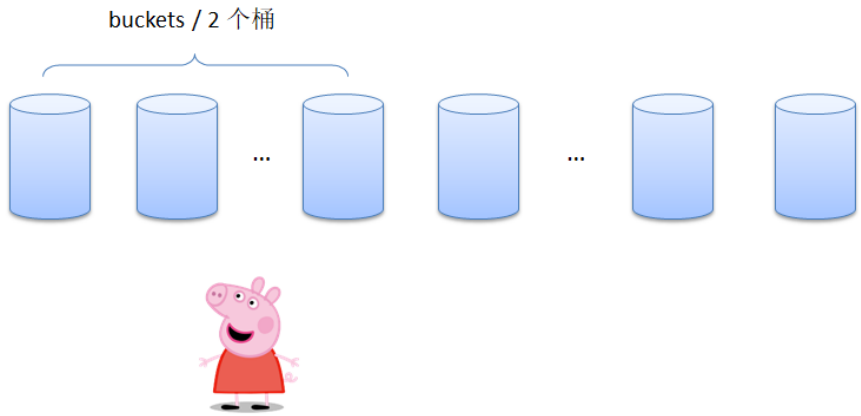
所以当前问题就是，给定 buckets 个桶，n 只小猪，最多进行 k 次实验，找出毒药桶。

（2）问题简化——只有一只小猪

我们可以先把问题简化一些，考虑只有一只小猪的情形。

容易想到，小猪一次喝一个桶的水，挨个喝过去，耗费 buckets 次实验，就一定能够找到毒药桶。所以能够完成的条件就是 k 大于等于 buckets 。

这样的搜索方式，显然类似遍历查找，耗费 O(n)的时间。为了提高效率，可以尝试让小猪一次喝多个桶的水，这样就可以快速排查了。如果我们一次排查一半的桶，这就相当于二分查找的思路，最终只需要 $\log_2$(buckets)次就可以找到了。



但是对于本题来说，小猪是无法一直查找下去的；如果中途遇到毒药桶，只能判断出毒药所在的大致范围。由于没有更多的小猪进一步排查，最坏情况下我们是得不到结果的。

于是可以有一个推测：需要的最少小猪数 n，可能就是对应做二分查找最坏情况下耗费的小猪数量。

比如考察 4 个桶，我们需要 $\log_2 4 = 2$ 只小猪，分 2 次实验。

但其实并不是这样。因为我们在这个过程中，限定了每次实验只能有一只小猪去喝水，这就导致想要得到结果，必然需要 $\log_2(buckets)$次实验，并且耗费 $\log_2(buckets)$只小猪。

那如果我们允许多只小猪在一次实验中同时喝水，又会怎么样呢？

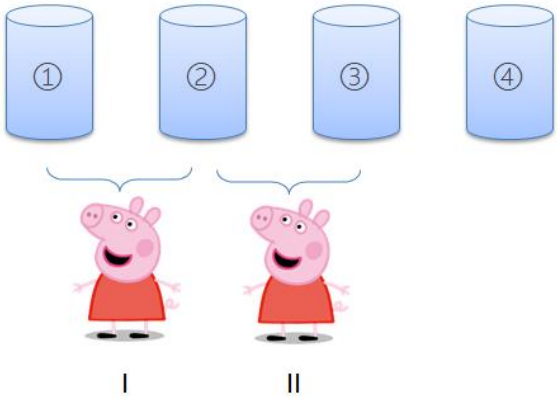
（3）问题简化——只有一次实验机会

我们依然简化条件，现在考虑 n 只小猪，但是只有一次实验机会。

最简单的考虑，当然还是每只小猪喝一个桶的水，这样只要有 n = buckets 只小猪，就可以找到毒药桶。

进一步考虑，小猪可以同时喝多个桶的水。不同的小猪交错开，喝不同桶的水，这样它们的组合情况，就有可能涵盖了所有的桶。

比如示例二中，4 个桶的一次实验中，我们只需要两只小猪就够了：



这样，如果小猪 I 牺牲，代表桶①②中存在毒药桶；如果幸存，则说明桶①②都不是毒药桶。小猪 II 同理。

最后的组合结果分析：

1. 小猪 I 牺牲，小猪 II 幸存：桶①是毒药桶
2. 小猪 I、II 都牺牲：桶②是毒药桶
3. 小猪 I 幸存，小猪 II 牺牲：桶③是毒药桶
4. 小猪 I、II 都幸存：桶④是毒药桶

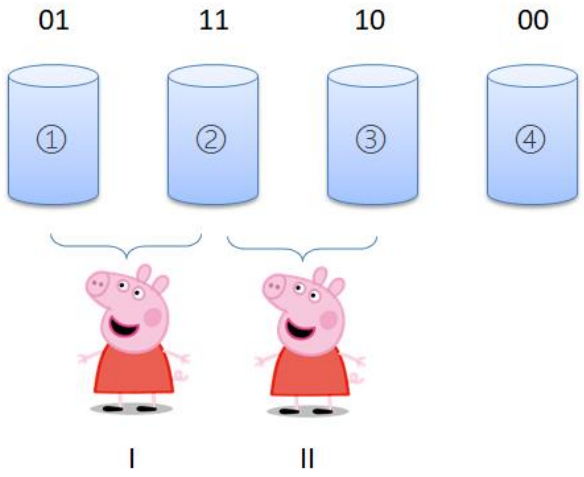
（4）问题抽象

我们把每个小猪看成一位数据，该位为 1 表示牺牲，为 0 表示幸存；那么 2 个二进制位就可以表示四个数据：

00（都幸存）、01（小猪 I 牺牲）、10（小猪 II 牺牲）、11（都牺牲）

因为小猪是否牺牲，跟喝了哪桶水有直接关系。所以我们可以给每种情形，对应一个桶是毒药桶：

- 00 号桶（小猪都幸存）：小猪 I、II 都没喝这个桶的水
- 01 号桶（小猪 I 牺牲、小猪 II 幸存）：小猪 I 喝了这个桶的水，小猪 II 没喝
- 10 号桶（小猪 I 幸存、小猪 II 牺牲）：小猪 II 喝了这个桶的水，小猪 I 没喝
- 11 号桶（小猪都牺牲）：小猪 I、II 都喝了这个桶的水



所以现在我们的问题，其实就是要用二进制编码所有的桶，最少需要的编码位数，就是我们要求的最少小猪数量。

可以想到，如果有 8 个桶，一次实验机会，我们只要 3 只小猪也就够了。对于一般情况，就是需要 $\log_2(buckets)$只小猪，一次实验直接搞定。

（5）问题扩展——多次实验

我们发现，上面的问题之所以能转换成二进制编码问题，关键就在于，每个小猪只有幸存和牺牲两种结果，于是就对应 0 和 1。只用 0 和 1 表示 1 位数据，当然就是二进制表示了。

如果现在可以进行多次（k 次）实验，小猪的结局就会更多：

- 幸存（0）
- 在第 1 次实验牺牲（1）
- 在第 2 次实验牺牲（2）
- …
- 在第 k-1 次实验牺牲（k-1）
- 在第 k 次实验牺牲（k）

我们类似地用 0~k 来表示这 k+1 种不同的结局，就相当于当前的一位（1 个 bit）可能的取值不仅仅是 0 和 1，而是 0~k。

这其实就是 k+1 进制的编码表达。

（6）举例说明

以 k = 2 为例，当前小猪有幸存、第 1 次实验牺牲、第 2 次实验牺牲三种结局，于是对应位可以有 0、1、2 三种取值。

考虑 n = 1，即只有一只小猪的情形。这其实就是最初我们考虑过的逐个桶尝试的方法。

对应地，我们可以将桶①、②、③分别编码为 1、2、0。通过小猪的结局状态，就可以直接推断出对应编码的桶是毒药桶。

接下来我们继续考虑 k = 2， n = 2 的场景。

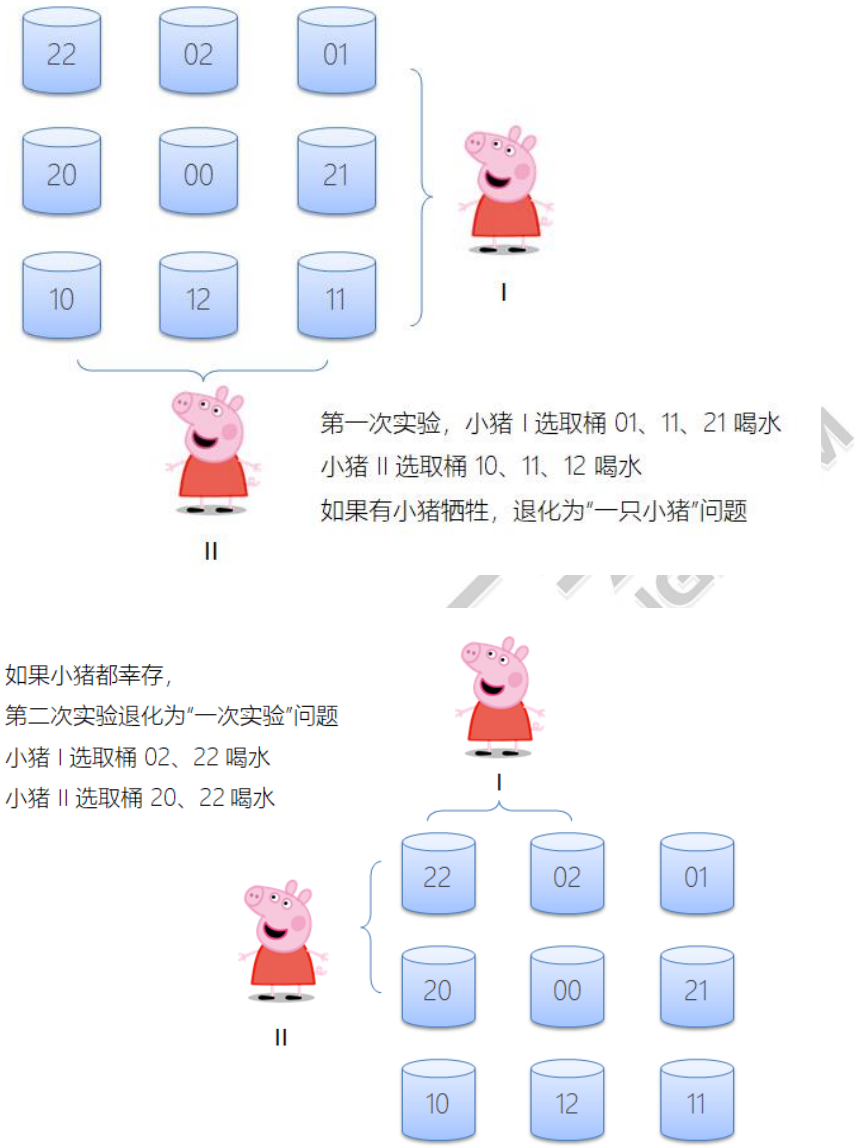
现在有两只小猪，它们各自都有三种结局状态，所以我们可以组合出 $3^2 = 9$ 种编码方式：

- 小猪 I、II 都幸存（00）
- 小猪 I 第一次实验牺牲，小猪 II 幸存（01）
- 小猪 I 第二次实验牺牲，小猪 II 幸存（02）
- 小猪 I 幸存，小猪 II 第一次实验牺牲（10）
- 小猪 I、II 都在第一次实验牺牲（11）
- 小猪 I 第二次实验牺牲，小猪 II 第一次实验牺牲（12）
- 小猪 I 幸存，小猪 II 第二次实验牺牲（20）
- 小猪 I 第一次实验牺牲，小猪 II 第二次实验牺牲（21）
- 小猪 I、II 都在第二次实验牺牲（22）

这其实就是**三进制数**的表示。

所以我们可以用 2 位最多表示出 9 个桶。桶的 2 位编码对应 2 个小猪：0 表示该小猪不喝这个桶的水；1 表示该小猪在第一次实验喝这桶水；2 表示该小猪在第二次实验喝这桶水。

我们进一步可以画出阶段图：



15.4.3 具体实现

我们总结可以发现，这其实就是一个编码问题。实验的次数 k，就对应着编码进制的基数 k+1；小猪数量 n，其实就是编码的码长。

对于一个 n 位的 k+1 进制数，它能表示的最大范围就是 $(k+1)^n$。

本题中为了能测试出结果，应该有

$$(k+1)^n \geq buckets$$

所以对应 n 的取值为

$$n \geq log_{k+1} buckets$$

最小值即为 $\log_{k+1}$buckets。

代码中为了方便计算对数，我们可以利用换底公式写作：

$$n_{min} = \frac{\log buckets}{\log(k+1)}$$

代码如下：

```
public class PoorPigs {

    public int poorPigs(int buckets, int minutesToDie, int minutesToTest)
{

        int k = minutesToTest / minutesToDie;

        return (int) Math.ceil(Math.log(buckets) / Math.log(k+1));

    }

}
```

复杂度分析

时间复杂度：O(1)。

空间复杂度：O(1)。