第五讲：四种算法思想

王争

前 Google 工程师

目录

贪心算法

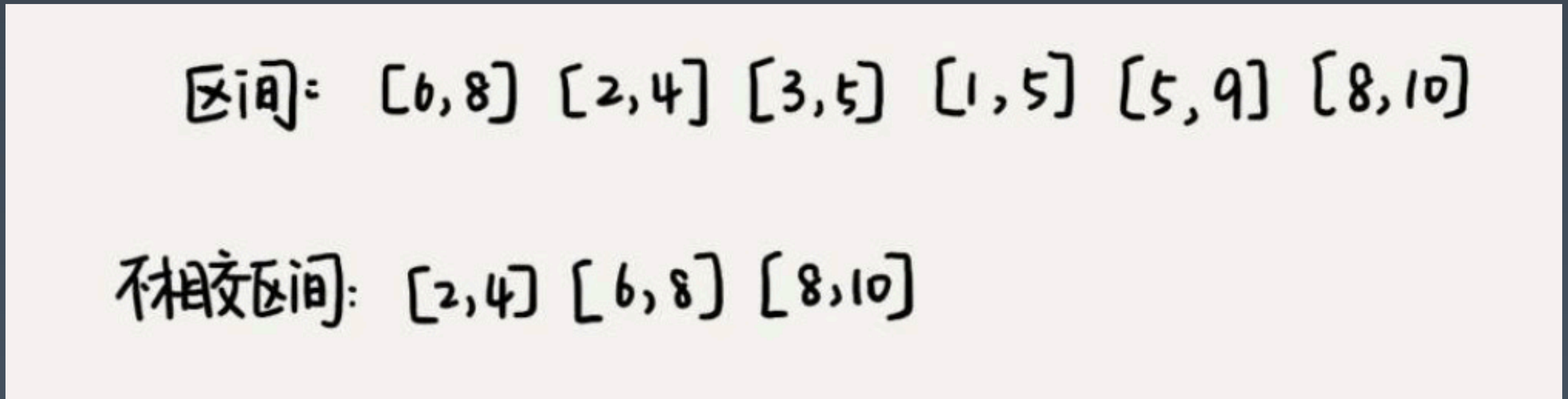
贪心算法举例分析

假设我们有 n 个区间，区间的起始端点和结束端点分别是：

[l1, r1]，[l2, r2]，[l3, r3]，……，[ln, rn]。

我们从这 n 个区间中选出一部分区间，这部分区间满足两两不相交（端点相交的情况不算相交）。

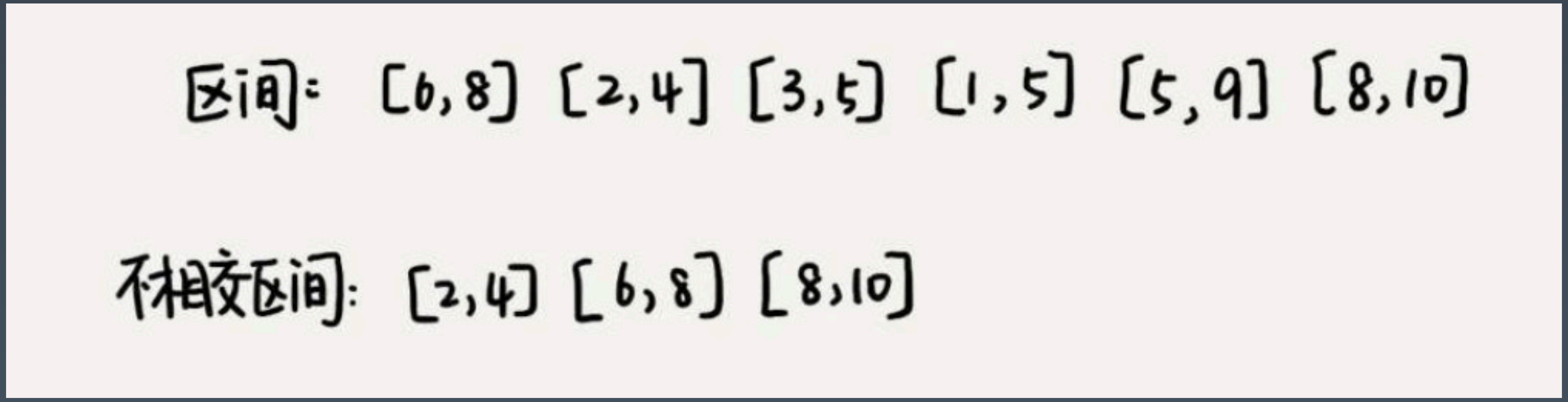
那么最多能选出多少个区间呢?



贪心算法举例分析

解决思路：我们假设这 n 个区间中最左端点是 lmin，最右端点是 rmax。这个问题就相当于，我们选择几个不相交的区间，从左到右将 [lmin, rmax] 覆盖上。

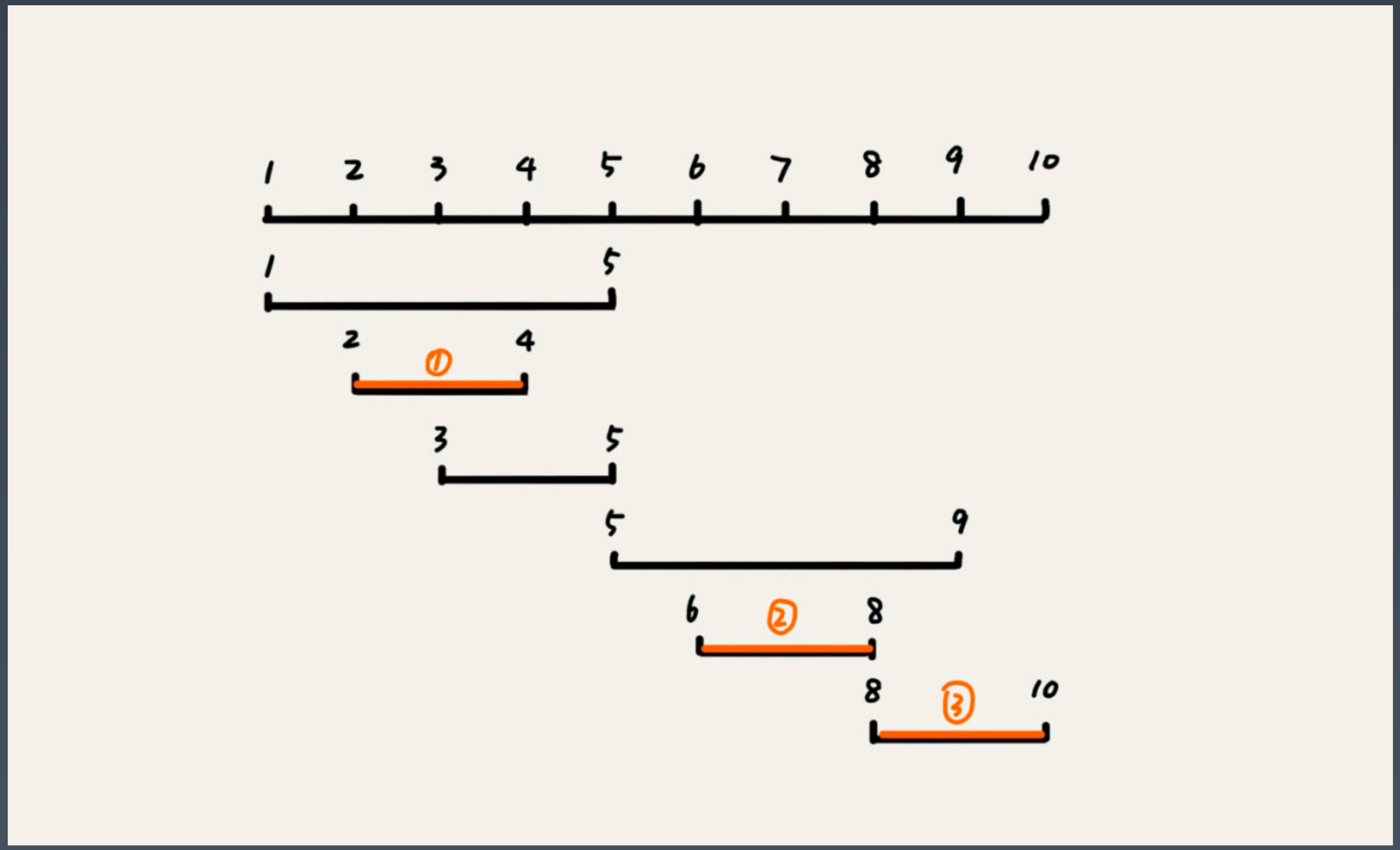
1. 我们按照起始端点从小到大的顺序对这 n 个区间排序。
2. 我们每次选择的时候，左端点跟前面的已经覆盖的区间不重合的，右端点又尽量小的，这样可以让剩下的未覆盖区间尽可能的大，就可以放置更多的区间。



贪心算法举例分析

区间: [6,8] [2,4] [3,5] [1,5] [5,9] [8,10]

不相交区间: [2,4] [6,8] [8,10]



贪心算法的难点

贪心算法正确性（最优性）证明：

大部分情况下，举几个例子验证一下就可以了。严格地证明贪心算法的正确性，是非常复杂的，需要涉及比较多的数学推理。而且，从实践的角度来说，大部分能用贪心算法解决的问题，贪心算法的正确性都是显而易见的，也不需要严格的数学推导证明。

分治算法

分治算法思想

1. 分治算法（divide and conquer）的核心思想：

分而治之， 也就是将原问题划分成 n 个规模较小，并且结构与原问题相似的子问题，递归地解决这些子问题，然后再合并其结果，就得到原问题的解。

2. 分治和递归的区别：

分治算法是一种处理问题的思想，递归是一种编程技巧。实际上，分治算法一般都比较适合用递归来实现。

3. 分治算法的递归实现中，每一层递归都会涉及这样三个操作：

分解：将原问题分解成一系列子问题；
解决：递归地求解各个子问题，若子问题足够小，则直接求解；
合并：将子问题的结果合并成原问题。

分治算法应用

1.指导编程和算法设计（降低时间复杂度）

- 如何编程求出一组数据的有序对个数或者逆序对个数呢?
- 二维平面上有 n 个点，如何快速计算出两个距离最近的点对?
- 有两个 n*n 的矩阵 A，B，如何快速求解两个矩阵的乘积 C=A*B?

2. 海量数据处理问题（内存放不下问题）

给 10GB 的订单文件按照金额排序这样一个需求，看似是一个简单的排序问题，但是因为数据量大，有 10GB，而我们的机器的内存可能只有 2～3GB，无法一次性加载到内存。

3. 并行计算（提高处理速度）

如果我们要处理的数据是 1T、10T、100T 这样的规模，那一台机器处理的效率肯定是非常低的。
一台机器过于低效，那我们就把任务拆分到多台机器上来处理。如果拆分之后的小任务之间互不干扰，独立计算，最后再将结果合并。

4. 指导架构、系统设计

MapReduce，对于谷歌搜索引擎来说，网页爬取、清洗、分析、分词、计算权重、倒排索引等等各个环节中，都会面对海量的数据（比如网页）。所以，利用集群并行处理显然是大势所趋。

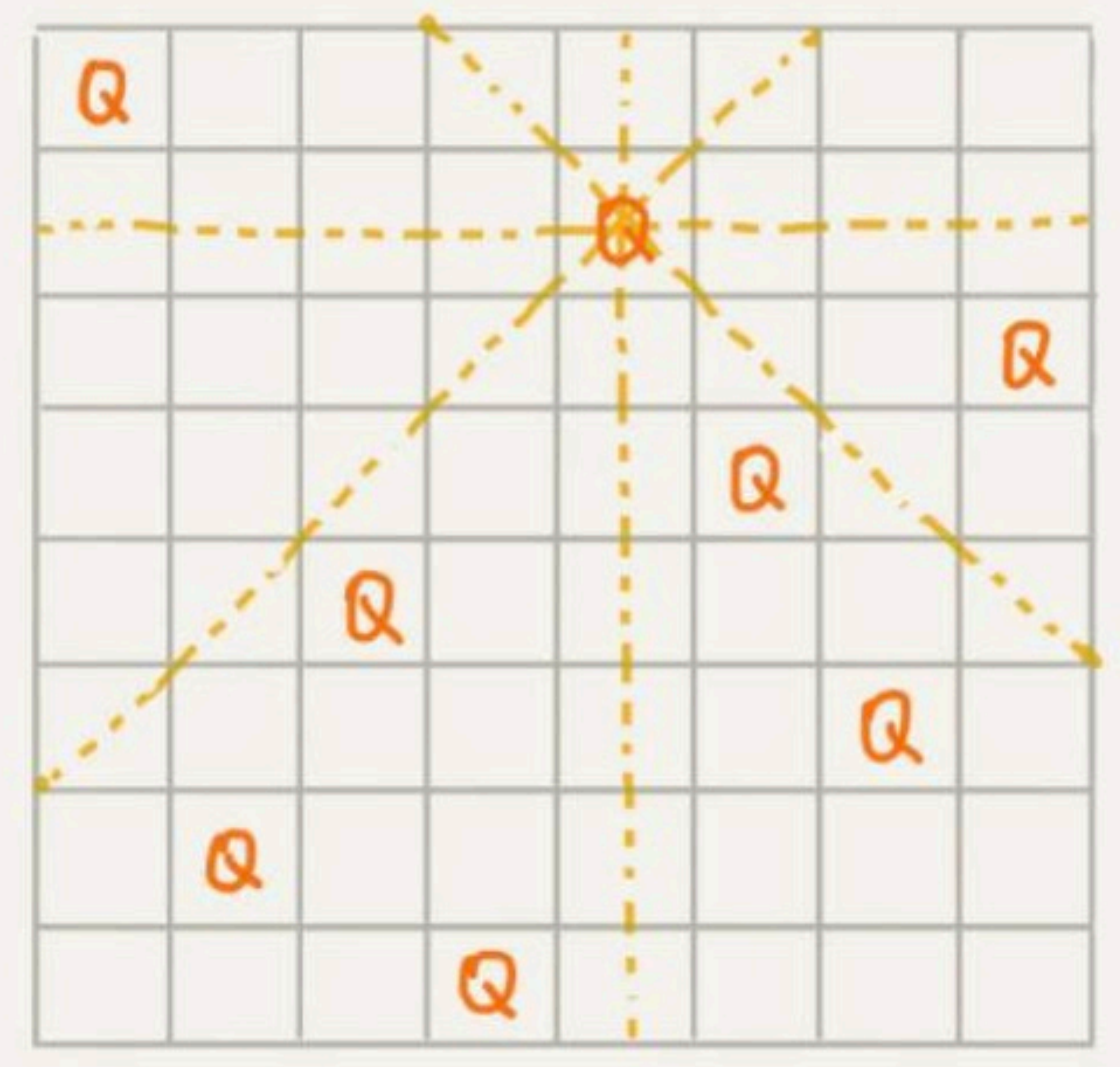
回溯算法

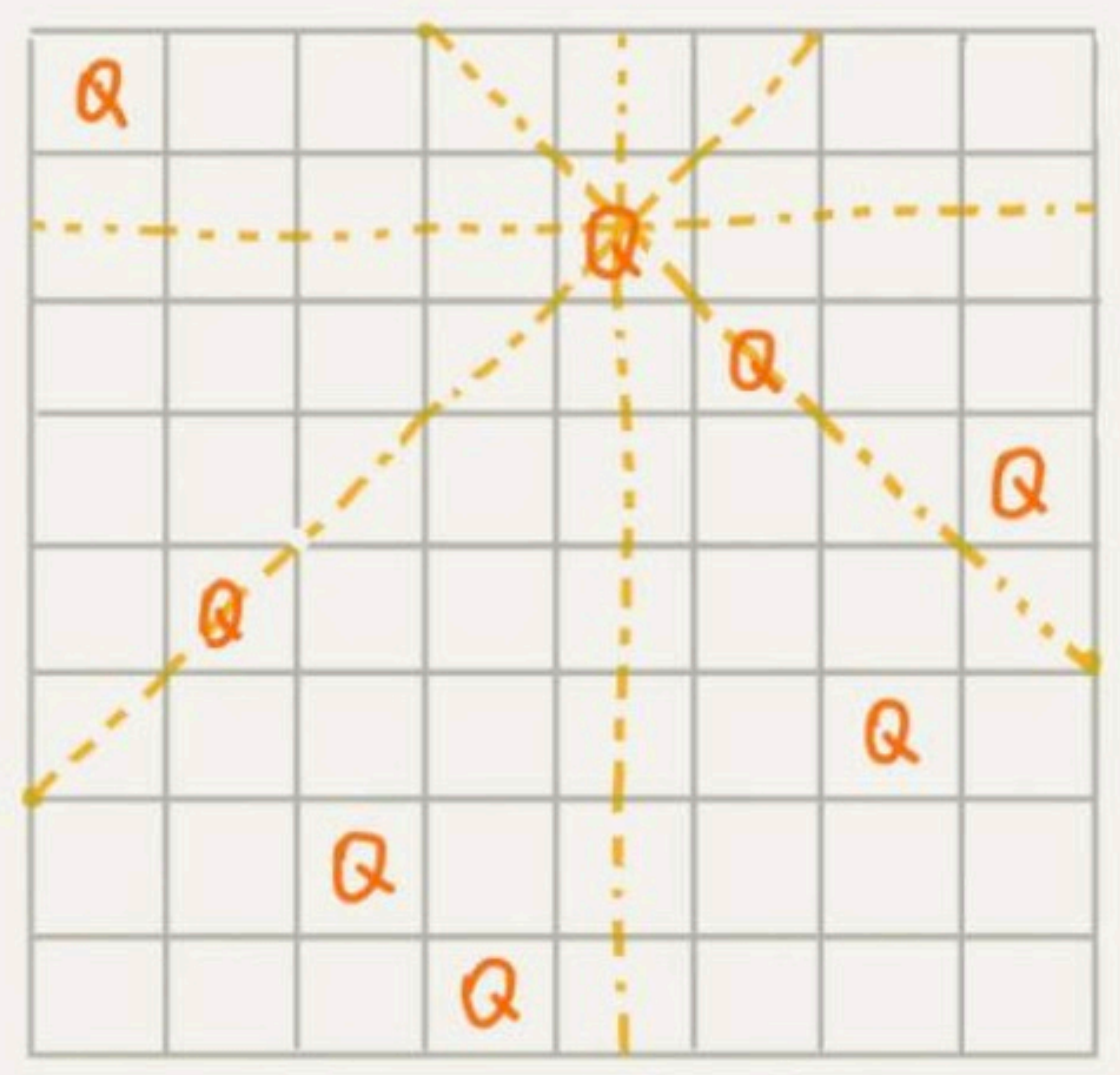
回溯算法思想

回溯的处理思想，有点类似枚举搜索。我们枚举所有的解，找到满足期望的解。为了有规律地枚举所有可能的解，避免遗漏和重复，我们把问题求解的过程分为多个阶段。每个阶段，我们都会面对一个岔路口，我们先随意选一条路走，当发现这条路走不通的时候（不符合期望的解），就回退到上一个岔路口，另选一种走法继续走。

回溯算法举例分析：八皇后问题

我们有一个 8x8 的棋盘，希望往里放 8 个棋子（皇后），每个棋子所在的行、列、对角线都不能有另一个棋子。你可以看我画的图，第一幅图是满足条件的一种方法，第二幅图是不满足条件的。八皇后问题就是期望找到所有满足这种要求的放棋子方式。



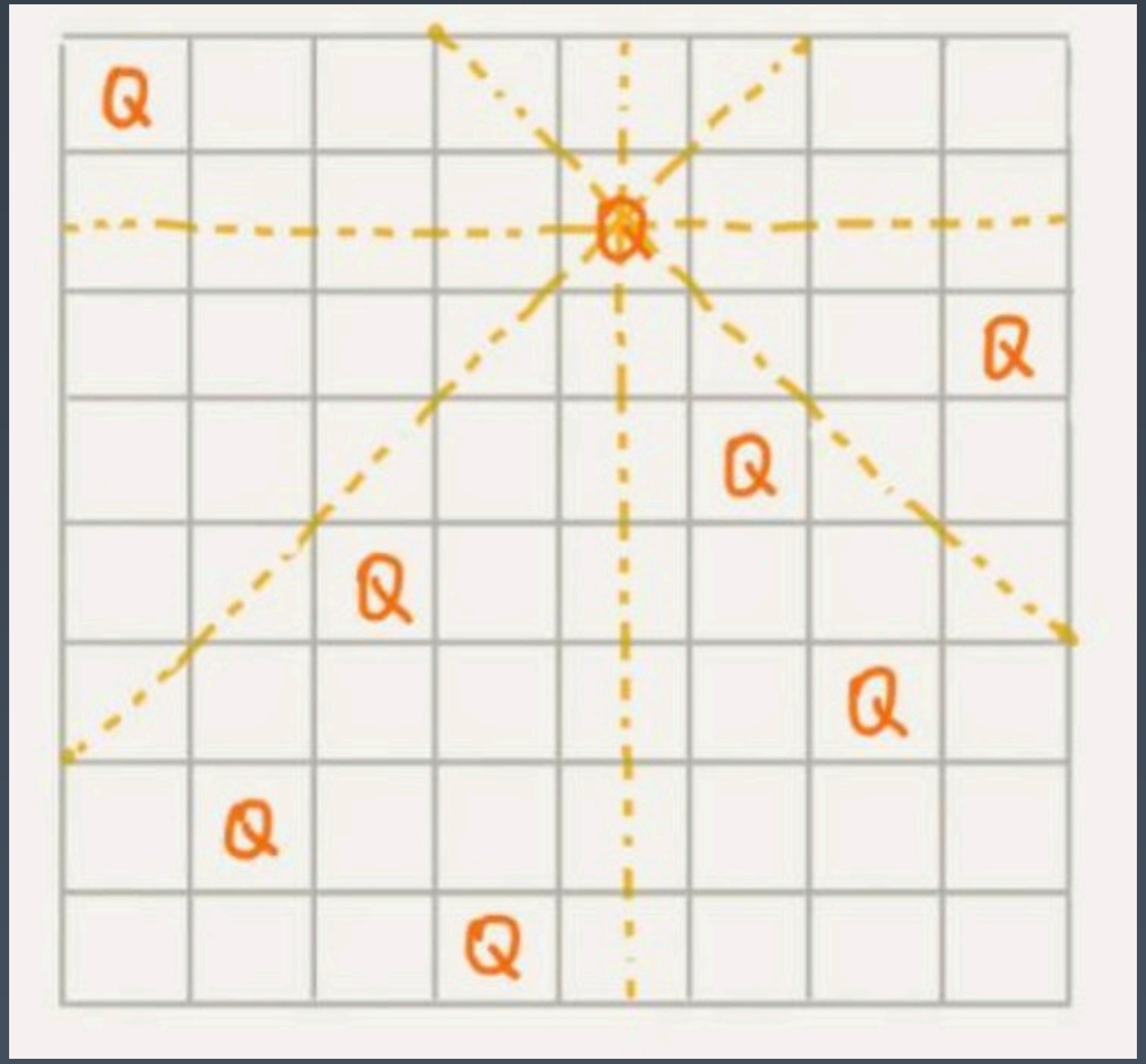


回溯算法举例分析：八皇后问题

我们把这个问题划分成8个阶段，依次将8个棋子放到第一行、第二行、第三行……第八行。在放置的过程中，我们不停地检查当前的方法，是否满足要求。如果满足，则跳到下一行继续放置棋子；如果不满足，那就再换一种方法，继续尝试。

```
int[] result = new int[8];

public void cal8queens(int row) {
  if (row == 8) {
    printQueens(result);
    return;
  }
  for (int column = 0; column < 8; ++column) {
    if (isOk(row, column)) {
      result[row] = column;
      cal8queens(row+1);
    }
  }
}
```



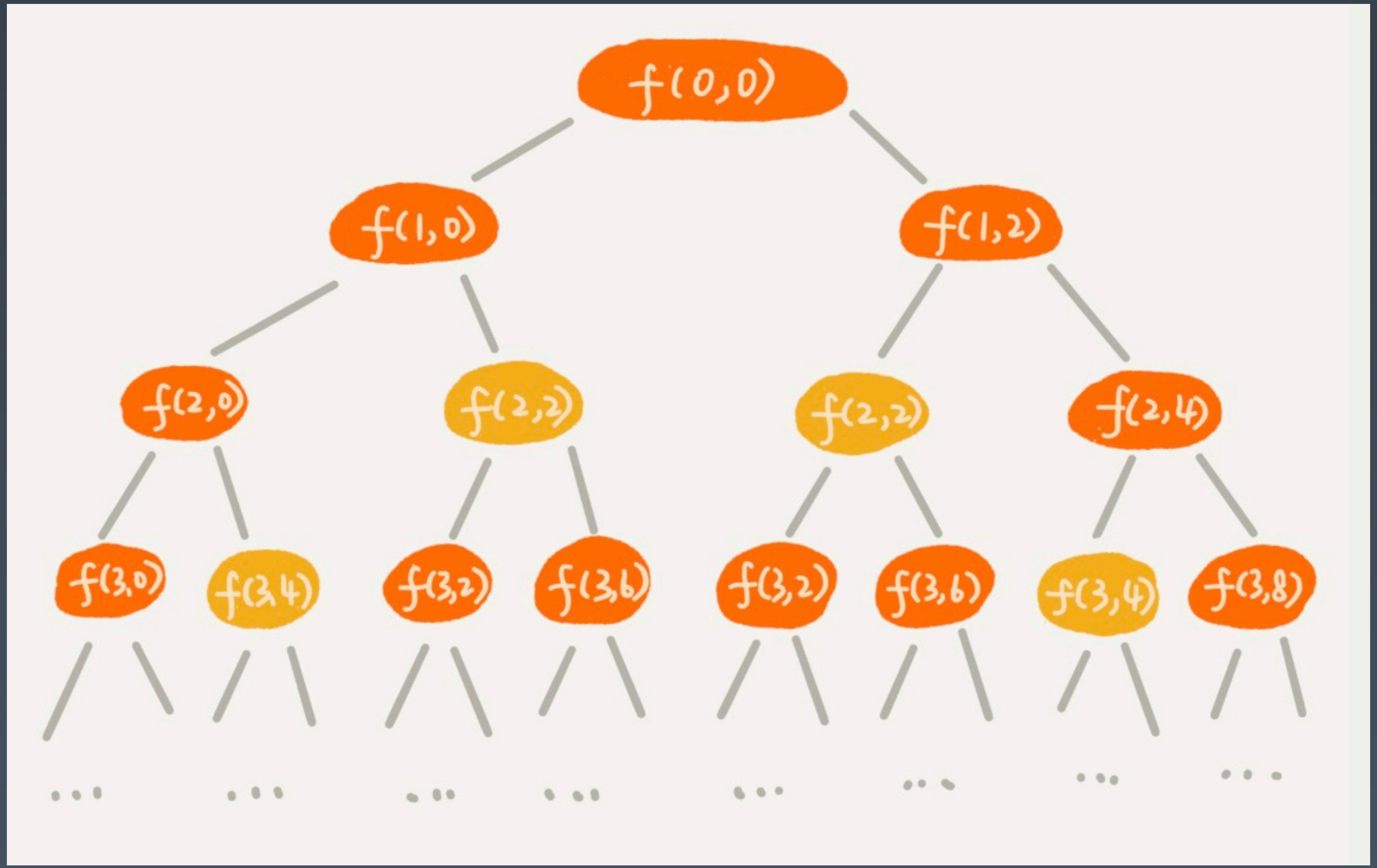
回溯算法举例分析：背包问题

0-1 背包问题有很多变体，我这里介绍一种比较基础的。我们有一个背包，背包总的承载重量是 W 千克。现在我们有 n 个物品，每个物品的重量不等，并且不可分割。我们现在期望选择几件物品，装载到背包中。在不超过背包所能装载重量的前提下，如何让背包中物品的总重量最大?

回溯算法举例分析-背包问题

我们可以把物品依次排列，整个问题就分解为了 n 个阶段，每个阶段对应一个物品怎么选择。先对第一个物品进行处理，选择装进去或者不装进去，然后再递归地处理剩下的物品。

```
// // 回溯算法实现。注意：我把输入的变量都定义成了成员变量。
private int maxW = Integer.MIN_VALUE; // 结果放到maxW中
private int[] weight = {2, ¼Œ2, ¼Œ4, ¼Œ6, ¼Œ3};  // 物品重量
private int n = 5; // 物品个数
private int w = 9; // 背包承受的最大重量
public void f(int i, int cw) { // 调用f(0, 0)
  // cw==w表示装满了，i==n表示物品都考察完了
  if (cw == w || i == n) {
    if (cw > maxW) maxW = cw;
    return;
  }
  f(i+1, cw); // 选择不装第i个物品
  if (cw + weight[i] <= w) {
    f(i+1,cw + weight[i]); // 选择装第i个物品
  }
}
```



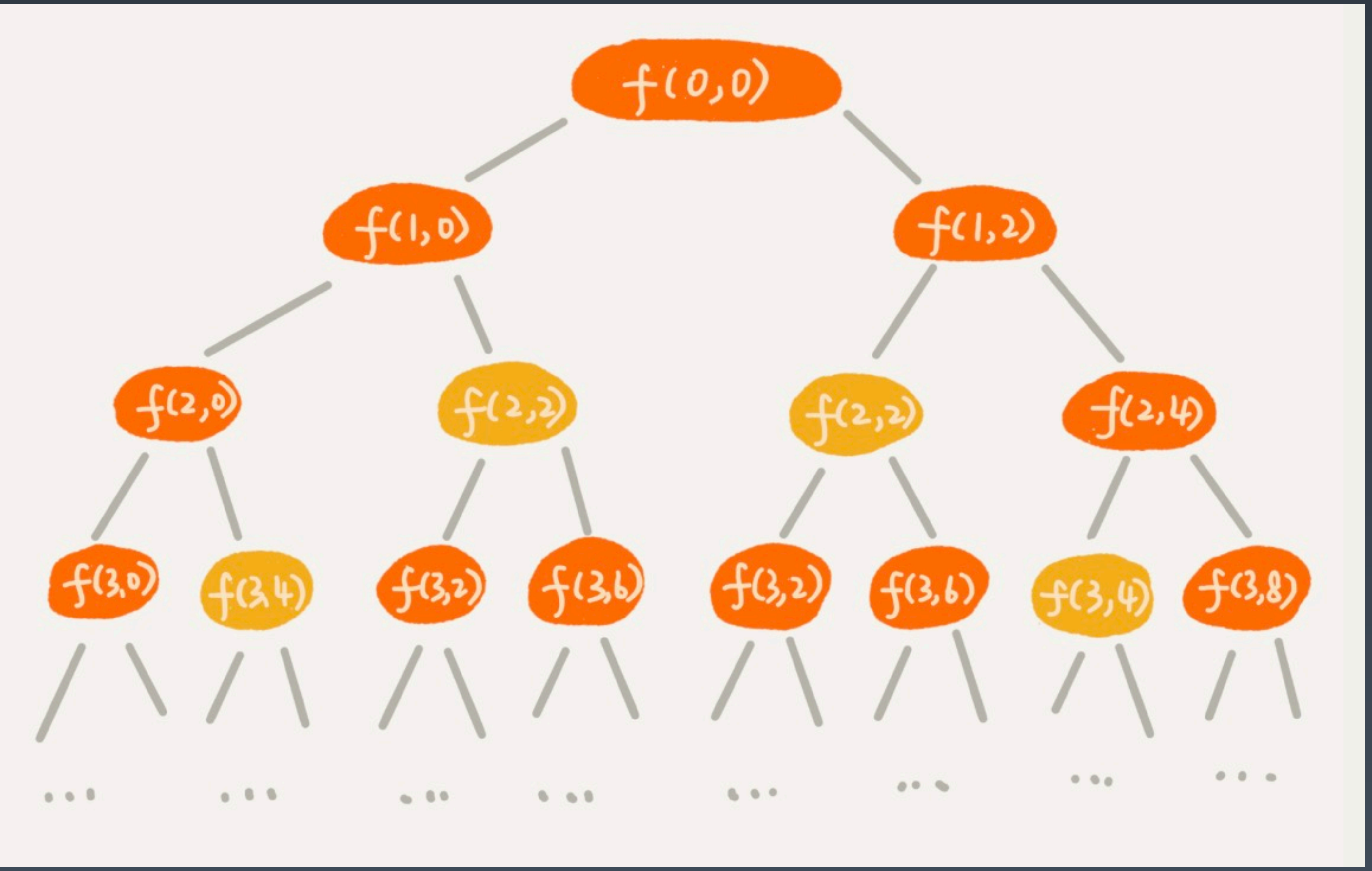
动态规划

0-1 背包问题的动态规划解法

我们有一个背包，背包总的承载重量是 W kg。现在我们有 n 个物品，每个物品的重量不等，并且不可分割。我们现在期望选择几件物品，装载到背包中。在不超过背包所能装载重量的前提下，如何让背包中物品的总重量最大?

重复子问题

```java
// 回溯算法实现。注意：我把输入的变量都定义成了成员变量。
private int maxW = Integer.MIN_VALUE; // 结果放到maxW中
private int[] weight = {2, 2, 4, 6, 3};  // 物品重量
private int n = 5; // 物品个数
private int w = 9; // 背包承受的最大重量
public void f(int i, int cw) { // 调用f(0, 0)
  // cw==w表示装满了，i==n表示物品都考察完了
  if (cw == w || i == n) {
    if (cw > maxW) maxW = cw;
    return;
  }
  f(i+1, cw); // 选择不装第i个物品
  if (cw + weight[i] <= w) {
    f(i+1,cw + weight[i]); // 选择装第i个物品
  }
}
```



一个模型，三个特征

多阶段决策最优解模型：

我们一般是用动态规划来解决最优问题。而解决问题的过程，需要经历多个决策阶段。每个决策阶段都对应着一组状态。然后我们寻找一组决策序列，经过这组决策序列，能够产生最终期望求解的最优值。

一个模型，三个特征

1.最优子结构

最优子结构指的是，问题的最优解包含子问题的最优解。反过来说就是，我们可以通过子问题的最优解，推导出问题的最优解。如果我们把最优子结构，对应到我们前面定义的动态规划问题模型上，那我们也可以理解为，后面阶段的状态可以通过前面阶段的状态推导出来。

2.无后效性

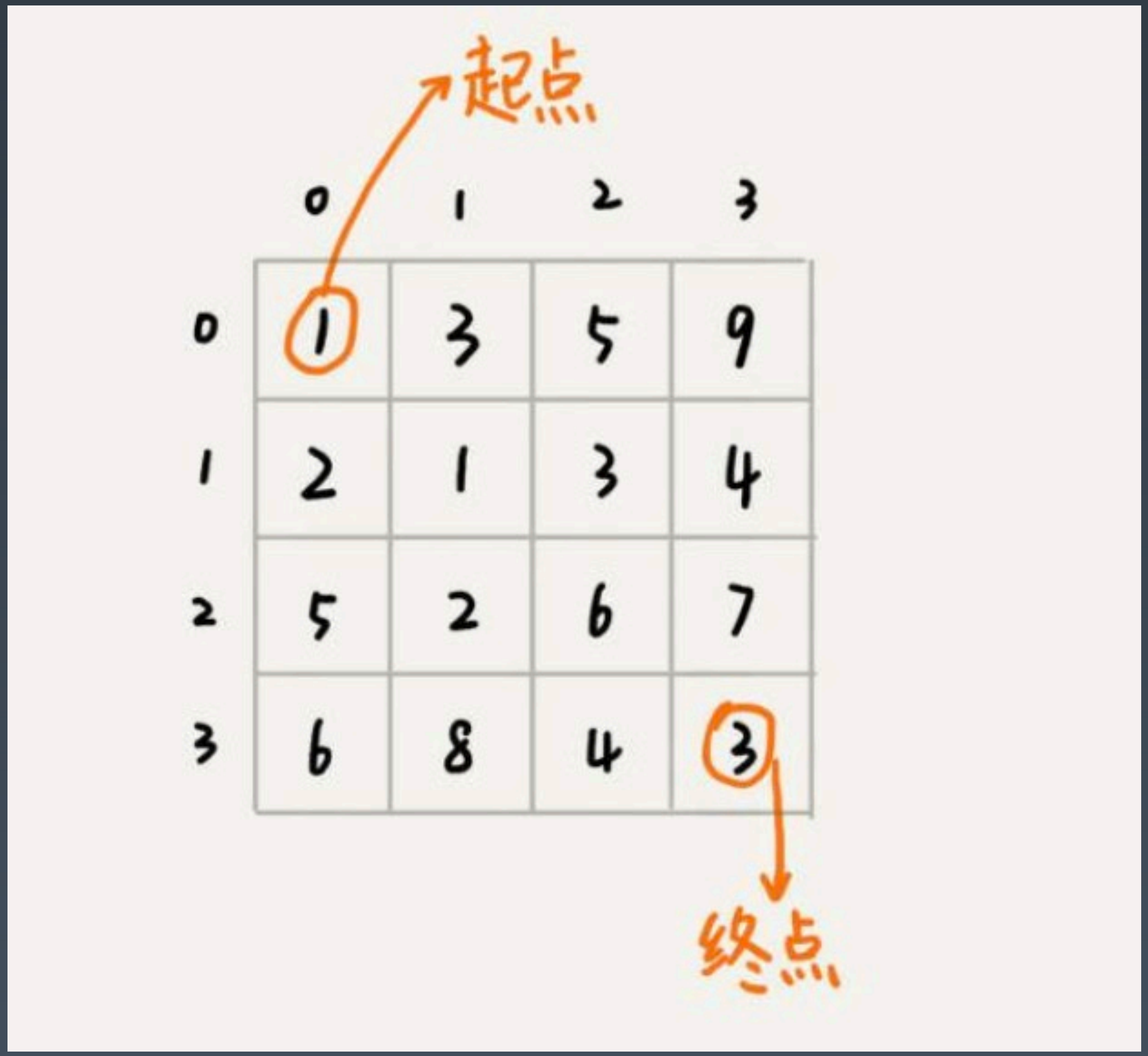
无后效性有两层含义，第一层含义是，在推导后面阶段的状态的时候，我们只关心前面阶段的状态值，不关心这个状态是怎么一步一步推导出来的。第二层含义是，某阶段状态一旦确定，就不受之后阶段的决策影响。无后效性是一个非常“宽松”的要求。只要满足前面提到的动态规划问题模型，其实基本上都会满足无后效性。

3.重复子问题

这个概念比较好理解。前面一节，我已经多次提过。如果用一句话概括一下，那就是：不同的决策序列，到达某个相同的阶段时，可能会产生重复的状态。

举例说明

假设我们有一个 n 乘以 n 的矩阵 w[n][n]。矩阵存储的都是正整数。棋子起始位置在左上角，终止位置在右下角。我们将棋子从左上角移动到右下角。每次只能向右或者向下移动一位。从左上角到右下角，会有很多不同的路径可以走。我们把每条路径经过的数字加起来看作路径的长度。那从左上角移动到右下角的最短路径长度是多少呢？



举例说明

我们先看看，这个问题是否符合“一个模型”?

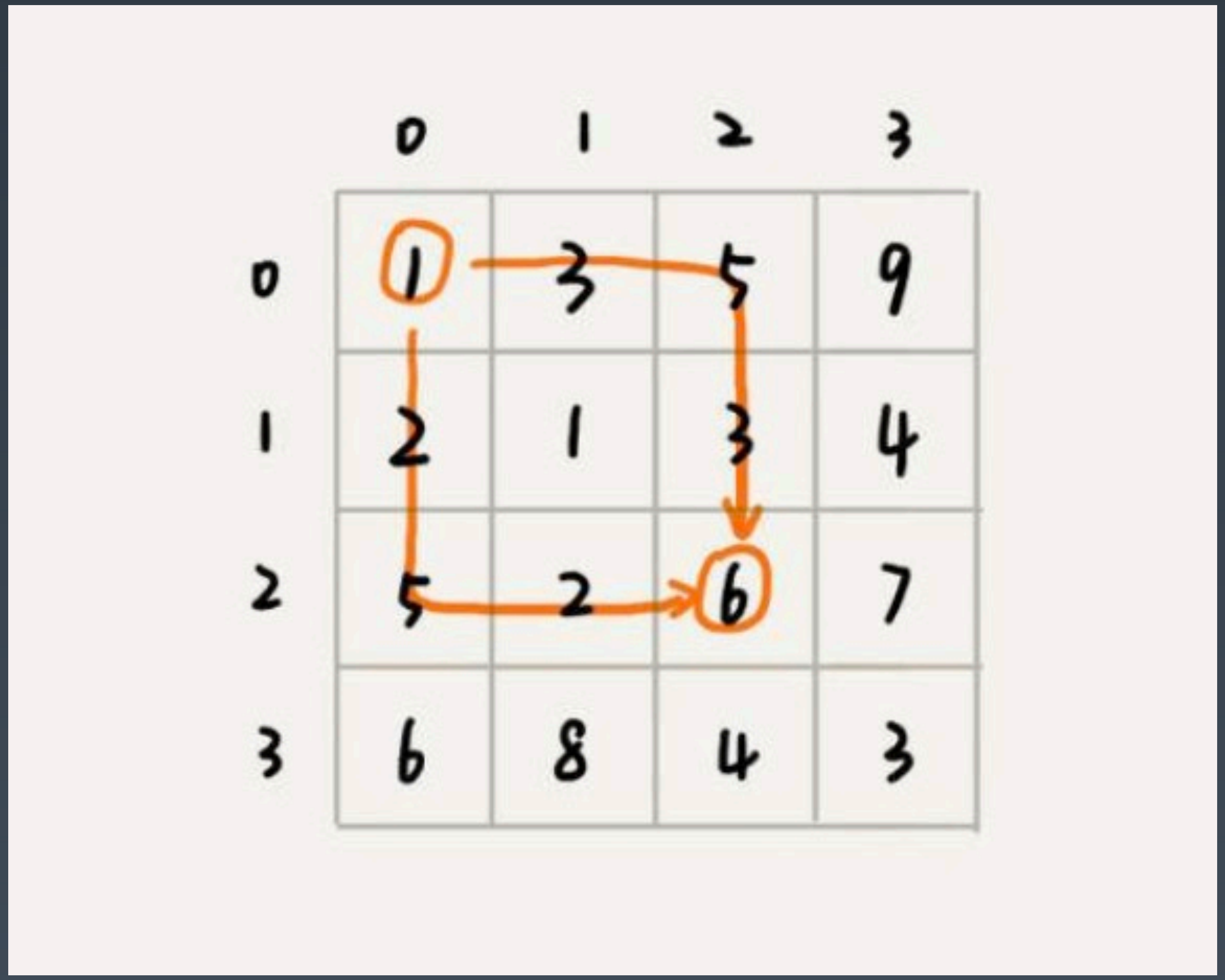
从 (0, 0) 走到 (n-1, n-1)，总共要走 2*(n-1) 步，也就对应着 2*(n-1) 个阶段。每个阶段都有向右走或者向下走两种决策，并且每个阶段都会对应一个状态集合。

我们把状态定义为 min_dist(i, j)，其中 i 表示行，j 表示列。min_dist 表达式的值表示从 (0, 0) 到达 (i, j) 的最短路径长度。所以，这个问题是一个多阶段决策最优解问题，符合动态规划的模型。

举例说明

重复子问题

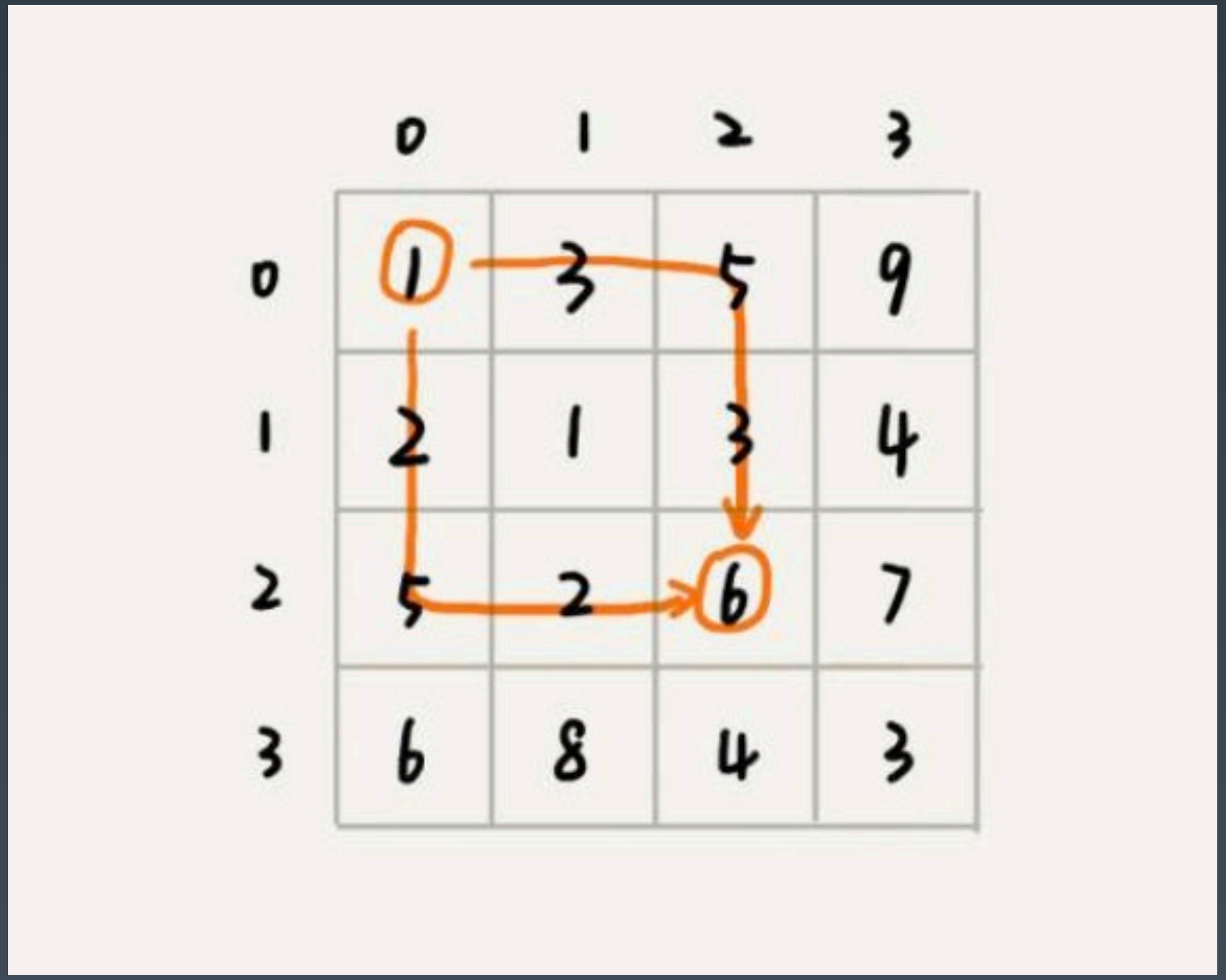
我们可以用回溯算法来解决这个问题。如果你自己写一下代码，画一下递归树，就会发现，递归树中有重复的节点。重复的节点表示，从左上角到节点对应的位置有多种路线，这也能说明这个问题中存在重复子问题。



举例说明

无后效性

如果我们走到 (i, j) 这个位置，我们只能通过 (i-1, j)，(i, j-1) 这两个位置移动过来，也就是说，我们想要计算 (i, j) 位置对应的状态，只需要关心 (i-1, j)，(i, j-1) 两个位置对应的状态，并不关心棋子是通过什么样的路线到达这两个位置的。而且，我们仅仅允许往下和往右移动，不允许后退，所以，前面阶段的状态确定之后，不会被后面阶段的决策所改变，所以，这个问题符合“无后效性”这一特征。

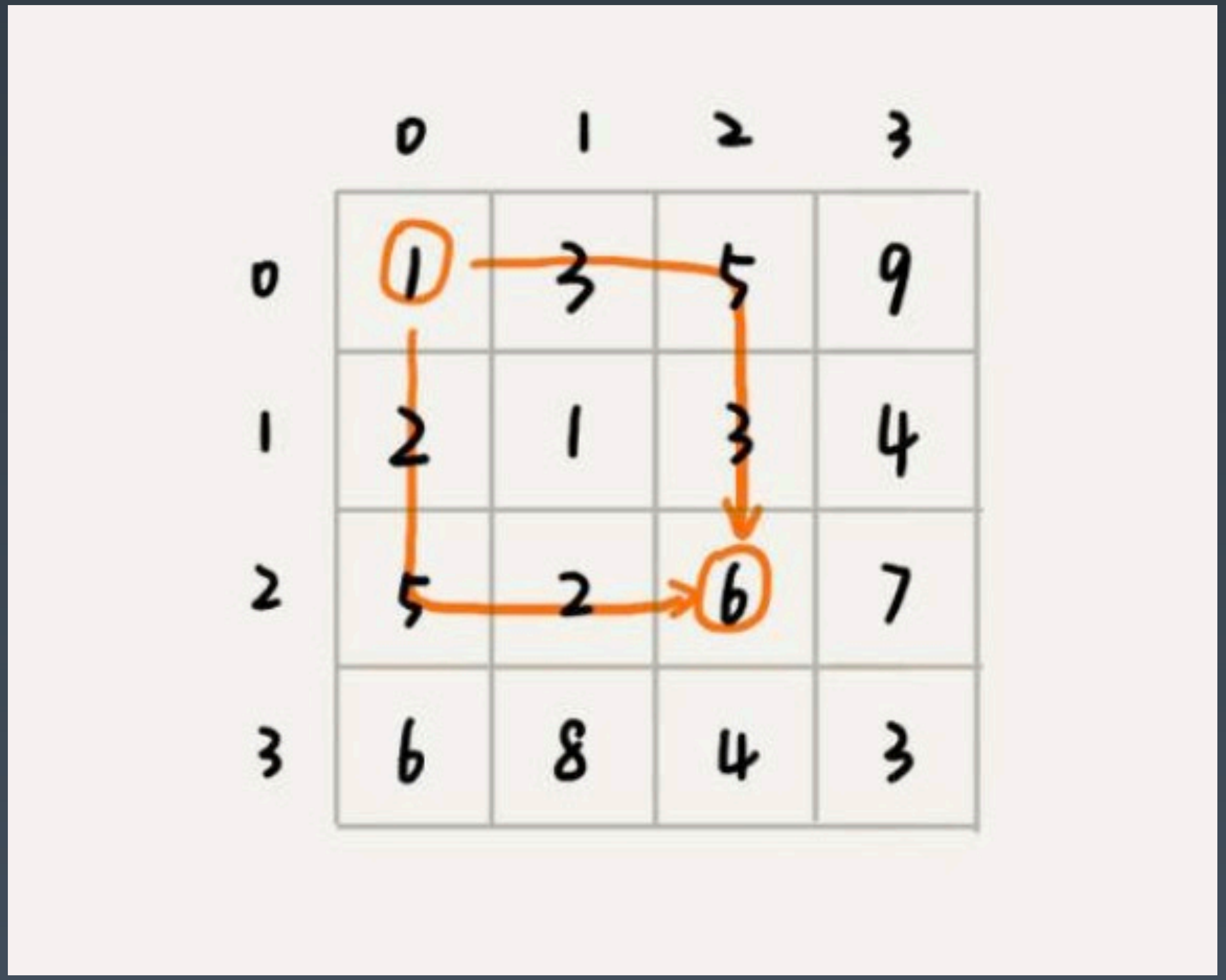


举例说明

最优子结构

刚刚定义状态的时候，我们把从起始位置 (0, 0) 到 (i, j) 的最小路径，记作 min_dist(i, j)。因为我们只能往右或往下移动，所以，我们只有可能从 (i, j-1) 或者 (i-1, j) 两个位置到达 (i, j)。也就是说，到达 (i, j) 的最短路径要么经过 (i, j-1)，要么经过 (i-1, j)，而且到达 (i, j) 的最短路径肯定包含到达这两个位置的最短路径之一。换句话说就是，min_dist(i, j) 可以通过 min_dist(i, j-1) 和 min_dist(i-1, j) 两个状态推导出来。这就说明，这个问题符合“最优子结构”。

min_dist(i, j) = w[i][j] + min(min_dist(i, j-1), min_dist(i-1, j))

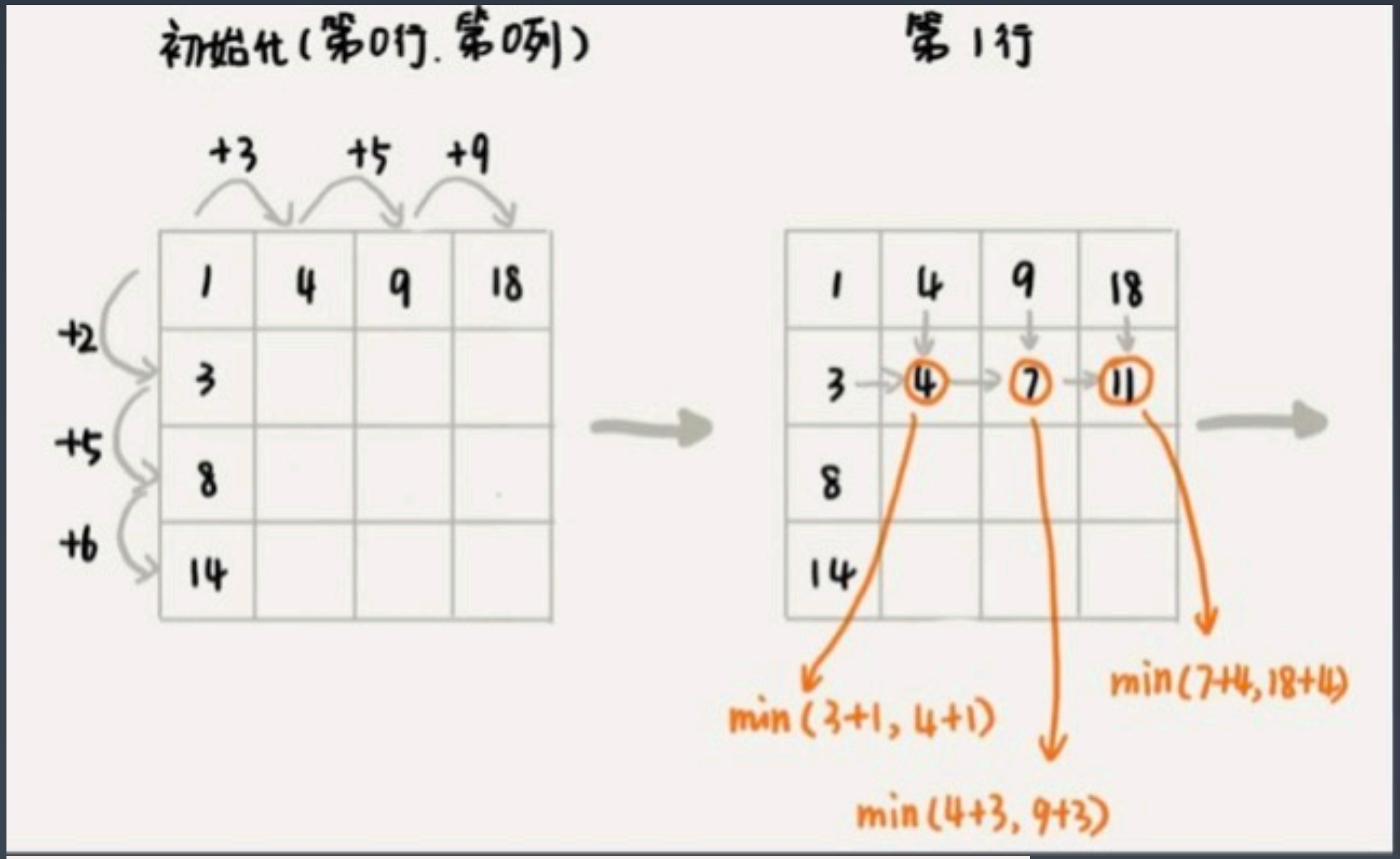


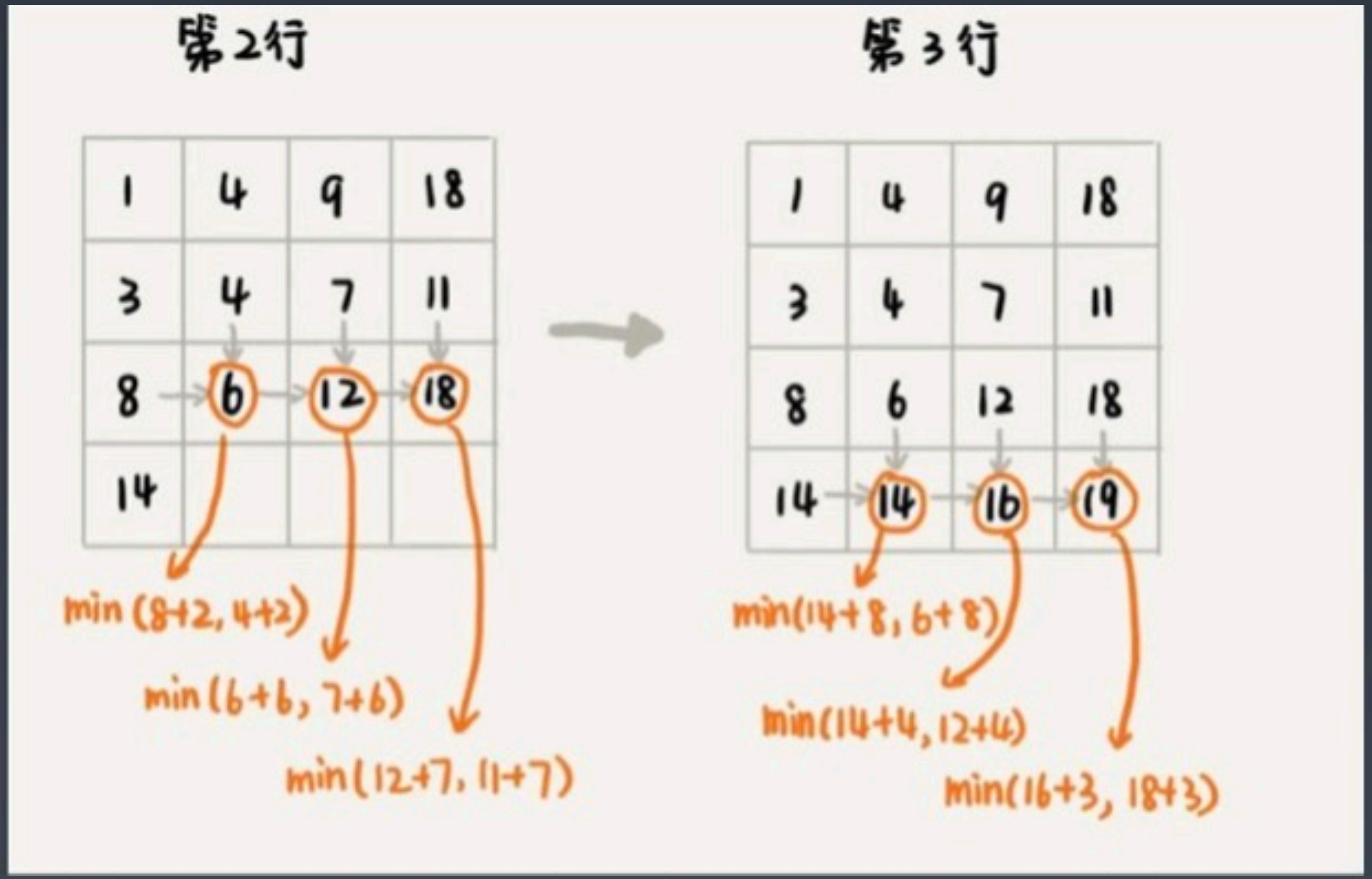
递归实现思路：基于备忘录

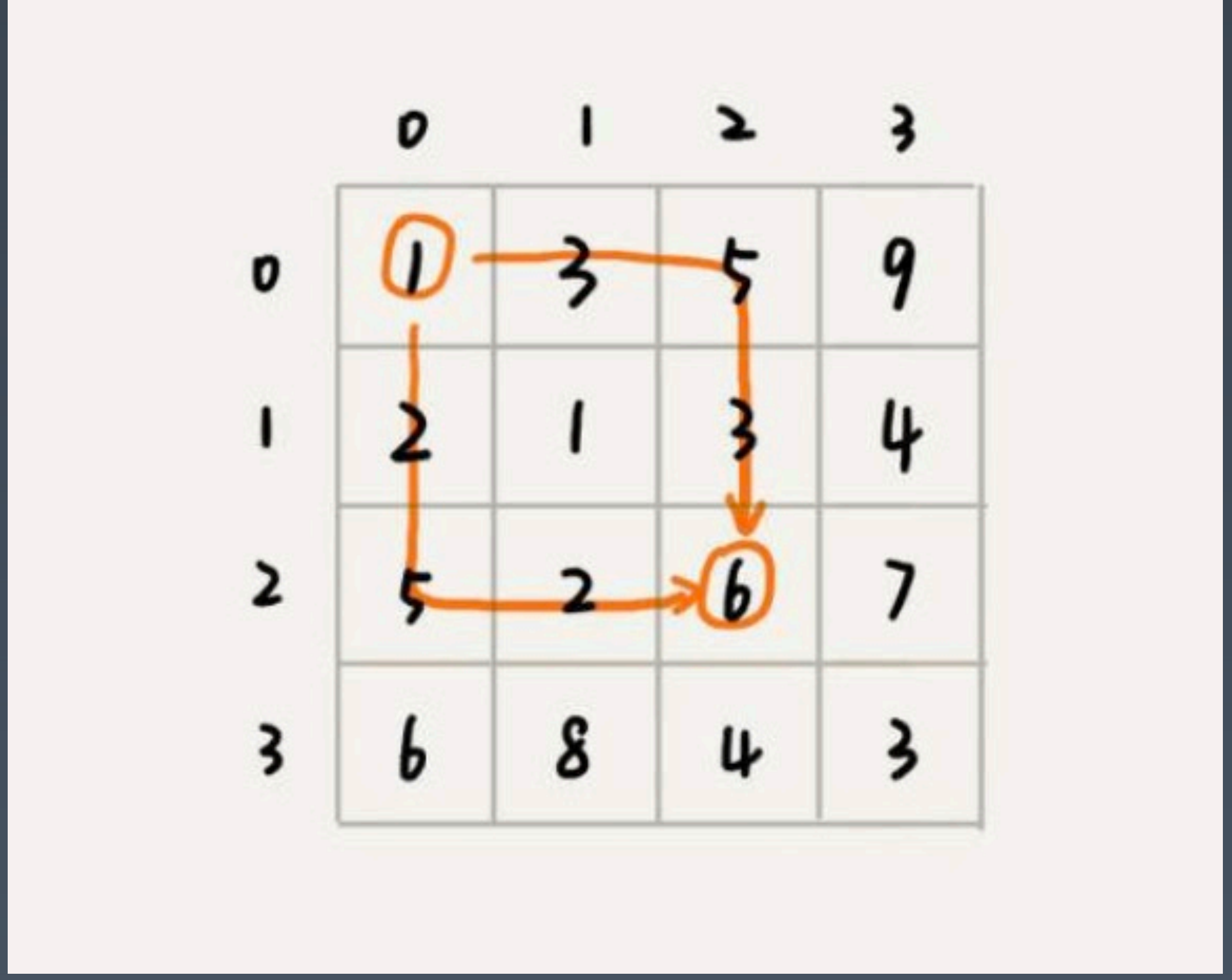
```java
private int[][] matrix =
        {{1, 3, 5, 9}, {2, 1, 3, 4}, {5, 2, 6, 7}, {6, 8, 4, 3}};
private int n = 4;
private int[][] mem = new int[4][44];
public int minDist(int i, int j) { // 调用minDist(n-1, n-1);
  if (i == 0 && j == 0) return matrix[0][0];
  if (mem[i][j] > 0) return mem[i][j];
  int minLeft = Integer.MAX_VALUE;
  if (j-1 >= 0) {
    minLeft = minDist(i, j-1);
  }
  int minUp = Integer.MAX_VALUE;
  if (i-1 >= 0) {
    minUp = minDist(i-1, j);
  }

  int currMinDist = matrix[i][j] + Math.min(minLeft, minUp);
  mem[i][j] = currMinDist;
  return currMinDist;
}
```

非递归实现思路：填表法







```
min_dist(i, j) = w[i][j] + min(min_dist(i, j-1), min_dist(i-1, j))
```

非递归实现思路

```java
public int minDistDP(int[][] matrix, int n) {
  int[][] states = new int[n][n];
  int sum = 0;
  for (int j = 0; j < n; ++j) { // 初始化states的第一行数据
    sum += matrix[0][j];
    states[0][j] = sum;
  }
  sum = 0;
  for (int i = 0; i < n; ++i) { // 初始化states的第一列数据
    sum += matrix[i][0];
    states[i][0] = sum;
  }
  for (int i = 1; i < n; ++i) {
    for (int j = 1; j < n; ++j) {
      states[i][j] =
            matrix[i][j] + Math.min(states[i][j-1], states[i-1][j]);
    }
  }
  return states[n-1][n-1];
}
```

THANKS!

极客大学