# 动态规划

## 简介

### 原理

|  |
| --- |
| 动态规划（Dynamic Programming，DP）是一种在数学和计算机科学中使用的，通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。  动态规划通常用于优化递归问题，例如斐波那契数列，如果运用递归的方式来求解会重复计算很多相同的子问题，利用动态规划的思想可以减少计算量。  动态规划的核心思想是记住已经解决过的子问题的解，这样就可以避免重复计算，从而提高效率。 |

### 适用范围

|  |
| --- |
| 动态规划是一种非常强大的算法，可以用来解决许多优化问题，如最短路径问题、最大子序列和等等。但是，动态规划也有其局限性，比如它不适合用于有无数个子问题需要解决的问题，或者子问题并不重叠的问题。在使用动态规划时，一定要确保问题满足“最优子结构”和“重叠子问题”和“无后效性”这两个条件。如果这两个条件得不到满足，那么问题就不能用动态规划来解决。  1、最优子结构：如果问题的一个最优解中包含了子问题的最优解，则该问题具有最优子结构。例如，你们学校有10个班，你已经计算出了每个班的最高考试成绩。那么现在我要求你计算全校最高的成绩，你会不会算？当然会，而且你不用重新遍历全校学生的分数进行比较，而是只要在这10个最高成绩中取最大的就是全校的最高成绩。  2、重叠子问题：用来求解原问题的递归算法反复地解同样的子问题，而不是总是在产生新的子问题。例如，动态规划会将每个求解过的子问题的解记录下来，这样当下一次碰到同样的子问题时，就可以直接使用之前记录的结果，而不是重复计算。  3、无后效性：一旦确定了某个状态，那么在这个状态之后的发展过程就不会再受到之前状态所产生的影响。也就是说，如果我们已经知道了到达某个状态的最优策略，那么在这个状态之后的决策不会影响到达这个状态时的最优策略。 |

### 步骤

|  |
| --- |
| 动态规划的步骤通常如下：  1、定义子问题：将原问题分解为子问题。子问题通常会有重叠，这意味着我们会多次遇到同样的子问题，只是在不同的问题中。  2、实现要反复执行来解决子问题的部分：这部分通常是一个简单的递归程序，或者是一个循环。  3、用数组保存子问题的解：子问题的解一旦计算出来就应该被保存，以便下次需要同样的子问题解的时候直接查表。  4、按照逻辑顺序，从小到大或者从大到小解决问题：有些问题可能从小到大解决会更容易，有些问题可能从大到小解决会更容易。  5、利用子问题的解，构建原问题的解：将子问题的解合并，得到原问题的解。 |

## 案例

## 类型

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **1. \*\*线性动态规划\*\*：**  - 描述：斐波那契数列问题，每个数等于前两个数和，求解第n个数。  - 转移方程：dp[i] = dp[i-1] + dp[i-2]。  - 转移方程含义：第i个数为前两个数之和。  **2. \*\*区间动态规划\*\*：**  - 描述：石子合并问题，n堆石子，第i堆有ai颗，每次选择相邻的两堆合并，代价是两堆石子和。求解全部合并为一堆的最小的合并代价。  - 转移方程：f[i][j] = min(f[i][j], f[i][k] + f[k + 1][j] + s[j] - s[i - 1])，其中 i <= k < j，s[i]是第1堆到第i堆的石子总数，s[j] - s[i - 1]即第i堆到第j堆的石子总数。  - 转移方程含义：从第i堆到第j堆石子的最小合并代价等于所有可能的分界线k的左区间代价+右区间代价+[i,j]区间石子额外代价的最小值。   |  | | --- | | 核心代码：  for l in range(1, n):  for i in range(1, n-l+1):  j = i + l  for k in range(i, j):  f[i][j] = min(f[i][j], f[i][k] + f[k+1][j] + s[j] - s[i-1]) |   **3. \*\*树形动态规划\*\*：**  - 描述：一家公司n个员工，除CEO外，每个员工都有一个直接上司。第i个员工有ai点快乐值，如果直接上司来了，TA就不来了。求最大快乐值。  - 转移方程：  # dp[i][0]代表i员工团队，TA不参加时的最大快乐值，dp[i][1]代表TA参加时的最大快乐值  dp[i][0] += max(dp[j][0], dp[j][1])  dp[i][1] += dp[j][0] # 其中 j 是 i 的子节点。  - 转移方程含义：i不参加的最大快乐为i下所有j参加和不参加的最大值的和；i参加的最大快乐为i的快乐值+所有j不参加的快乐值。   |  | | --- | | 核心代码：  def dfs(i):  dp[i][1] = happy[i]  for j in edges[i]:  dfs(j)  dp[i][0] += max(dp[j][0], dp[j][1])  dp[i][1] += dp[j][0] |   **4. \*\*背包动态规划\*\*：**  - 描述：0-1背包问题，给定n物品，每个物品有重量Wi和价值Vi。求一个容量m的背包能装入的最大价值。  - 转移方程：dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]] + v[i])，其中 w[i] 和 v[i] 分别是第 i 个物品的重量和价值。  - 转移方程含义：装到i物品和j容量时的最大价值为：装前一个物品和不装前一个物品时扣除重量来装当前物品，两者价值的最大值，前提是容量够装当前物品。   |  | | --- | | 核心代码示例：  for num in range(1, 1 + len(main)):  for mon in range(1 + money):  dp[num][mon] = dp[num - 1][mon] # 当前物品继承上一个物品的最大价值  (p1, v1), (p2, v2), (p3, v3) = main[num - 1]  # 不买这个物品（买前一个物品） 和 不买前一个物品买这个物品的最大值  if mon >= p1 + p2 + p3:  # 这里必须是dp[num][mon]不能是dp[num-1][mon]，因为这个值是在每个if更新的  dp[num][mon] = max(dp[num][mon], dp[num - 1][mon - p1 - p2 - p3] + v1 + v2 + v3)  if mon >= p1 + p3:  dp[num][mon] = max(dp[num][mon], dp[num - 1][mon - p1 - p3] + v1 + v3)  if mon >= p1 + p2:  dp[num][mon] = max(dp[num][mon], dp[num - 1][mon - p1 - p2] + v1 + v2)  if mon >= p1:  dp[num][mon] = max(dp[num][mon], dp[num - 1][mon - p1] + v1) |   **5. \*\*坐标动态规划\*\*：**  - 描述：请计算n\*m的棋盘格子，从左上到右下共有多少种走法。  - 转移方程：dp[i][j] = dp[i - 1][j] + dp[i][j - 1] if i != 0 and j != 0 else 1  - 转移方程含义：到达位置(i, j)的方法数为到达上方的方法数+到达左方的方法数和  **6. \*\*字符串动态规划\*\*：**  - 描述：最长回文子串问题，求解一个字符串的最长回文子串。  - 转移方程：dp[i][j] = (s[i] == s[j]) and dp[i+1][j-1]，其中 s[i] 和 s[j] 是字符串的字符。  - 转移方程含义：字符串从第i个字符到第j个字符是否为回文等于第i个字符是否等于第j个字符和字符串从第i+1个字符到第j-1个字符是否为回文的逻辑与。   |  | | --- | | 核心代码示例：  # 需先求长度为1的各个位置是否回文，再求长度为2的各个位置是否回文  # 枚举子串的长度 l+1  for l in range(n):  # 枚举子串的起始位置 i，这样可以通过 i+l 得到子串的结束位置  for i in range(n):  j = i + l  if j >= len(s): # 如果子串的结束位置超过字符串的长度，就跳出循环  break  if l == 0: # 如果子串的长度为1，那么它一定是回文  dp[i][j] = True  elif l == 1: # 如果子串的长度为2，那么只有当两个字符相等时，它才是回文  dp[i][j] = (s[i] == s[j])  else: # 如果子串的长度大于2，那么只有当首尾字符相等，并且去掉首尾的子串是回文时，它才是回文  dp[i][j] = (dp[i + 1][j - 1] and s[i] == s[j])  if dp[i][j] and l + 1 > len(ans): # 如果子串是回文，并且长度大于当前最长回文子串的长度，就更新最长回文子串  ans = s[i:j+1] |   **7. \*\*判定型动态规划\*\*：**  - 描述：子集和问题，判断是否存在一个子集的和等于给定值。  - 转移方程：dp[i][j] = dp[i-1][j] or dp[i-1][j-nums[i]]，其中 nums[i] 是数组中的数。  - 转移方程含义：dp:前i个数是否存在和为j的子集: 等于前i-1个数是否存在和为j的子集 或者 前i-1个数是否存在和为j-nums[i] 的子集。   |  | | --- | | 核心代码：  for i in range(1, n + 1):  for j in range(1, target + 1):  if j < nums[i - 1]: # 当前目标和小于当前元素时，只能选择不取  dp[i][j] = dp[i - 1][j]  else: # 当前目标和大于等于当前元素时，可以选择取或者不取  dp[i][j] = dp[i - 1][j] or dp[i - 1][j - nums[i - 1]] |   **8. \*\*目标型动态规划\*\*：**  - 描述：最大子数组和问题，求解一个带负数的数组的最大子数组和。  - 转移方程：dp[i] = max(dp[i-1] + nums[i], nums[i])，其中 nums[i] 是数组中的数。  - 转移方程含义：前i个数的最大子数组和等于前i-1个数的最大子数组和加上第i个数和第i个数的最大值。   |  | | --- | | 核心代码：  for i in range(1, n):  # 状态转移方程，dp[i] 表示以第 i 个元素结尾的最大子数组和  dp[i] = max(dp[i - 1] + nums[i], nums[i])  # 更新最大子数组和  max\_sum = max(max\_sum, dp[i]) |   **9. \*\*概率动态规划\*\*：**  - 描述：在N个格子的洞穴中，每个格子有若干黄金。初始在1位置，投掷6面骰子决定往前走几步。求到N格子时期望黄金数。（新位置在N外当无效）  - 转移方程：expected\_gold[i] = gold[i] + total\_expected\_gold / min(6, n - i - 1)  - 转移方程含义：在第i个格子的期望黄金数=往后6个(或者更少)格子的期望黄金数的均值   |  | | --- | | 代码：  def solve():  # 3  # 3 6 9  # 输出15  # 读取洞穴的维度  n = int(input().strip())  # 读取每个单元格的黄金数量  gold = list(map(int, input().strip().split()))  # 初始化期望的黄金为每个单元格的黄金  expected\_gold = gold[:]  # 从洞穴的末尾到开始进行迭代  for i in range(n - 2, -1, -1):  # 位置i往后走能拿到的总的黄金数  total\_expected\_gold = 0  # 往后走1到6步，接近洞口则最多只能走n-i-1步  for j in range(1, min(7, n - i)):  total\_expected\_gold += expected\_gold[i + j]  # 当前位置期望的黄金数为后面步数总的期望的黄金数的平均值  expected\_gold[i] = gold[i] + total\_expected\_gold / min(6, n - i - 1)  # 打印第一个单元格的期望黄金  print('%.10f' % expected\_gold[0]) |   **10. \*\*状压动态规划\*\*：**  - 描述：旅行商问题（TSP），求解访问所有城市并返回原点的最短路径。  - 转移方程：dp[i][j] = min(dp[i][j], dp[k][j|(1<<i)] + dis[k][i])，其中 dis[k][i] 是城市 k 和城市 i 之间的距离。  - 转移方程含义：访问过的城市集合为j，当前在城市i的最短路径长度等于所有可能的上一个城市k的最短路径长度和城市k到城市i的距离的最小值。  **11. \*\*数位动态规划\*\*：**  - 描述：数位统计问题，求解一个数字中包含的特定数字的个数。  - 转移方程：dp[i][j] = dp[i-1][j] + dp[i-1][j-1]，其中 j 是数字的位数。  - 转移方程含义：前i位中j位为1的数的个数等于前i-1位中j位为1的数的个数和前i-1位中j-1位为1的数的个数的和。 |

## 解题思路

|  |
| --- |
| 1、定义状态：  首先，我们需要定义一个或多个状态，这些状态能够反映问题的某种性质。例如，在背包问题中，我们可以定义状态 dp[i][j] 为前 i 个物品，总重量不超过 j 的情况下的最大价值。  2、找出状态转移方程：  状态转移方程描述了状态之间是如何转移的。在背包问题中，状态转移方程可以是 dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]] + v[i])，其中 w[i] 和 v[i] 分别是第 i 个物品的重量和价值。  2.1、**首先要找到所有「状态」和每个状态可以做的「选择」，然后择优。**  3、确定初始状态和边界条件：  初始状态和边界条件是解决动态规划问题的关键。在背包问题中，初始状态可以是 dp[0][j] = 0（没有物品时的价值为0），边界条件可以是 j 必须大于等于 w[i]。 |

# 深度优先搜索

## 简介

### 原理

|  |
| --- |
| 深度优先搜索的基本思想是从图的某个顶点开始，沿着一条路径尽可能深入地搜索，直到这条路径已经被完全探索，然后回溯并沿着未探索的路径继续搜索，直到所有的顶点都被访问过。这种方法类似于在迷宫中寻找出口，总是沿着未探索的路径前进，直到无路可走时才回头。 |

### 适用范围

|  |
| --- |
| 深度优先搜索适用于许多问题，包括寻找图的连通分量、检测图中是否存在环、寻找图的拓扑排序、解决迷宫问题等。此外，深度优先搜索也常用于树的遍历，如二叉树的前序、中序和后序遍历。 |

### 步骤

|  |
| --- |
| 选择起始节点：选择一个起始节点作为搜索的起点。  访问节点：访问当前节点，并将其标记为已访问。  深入搜索：从当前节点的所有未访问过的邻居中选择一个，然后对该邻居节点进行深度优先搜索。这个步骤是递归的，也就是说，我们会持续深入搜索，直到找到一个没有未访问过的邻居的节点。  回溯：如果当前节点没有未访问过的邻居，或者所有邻居都已经被访问过，那么回溯到当前节点的上一个节点（也就是当前节点的父节点），并在那里继续搜索。  结束：当所有可达的节点都已经被访问过，搜索结束。 |

## 案例

|  |
| --- |
|  |

## 解题思路

|  |
| --- |
| 理解问题：首先，你需要理解问题的需求和限制条件。这将帮助你确定是否应该使用深度优先搜索，以及如何定义搜索的目标和约束。  定义搜索空间：确定你需要搜索的空间。这通常是一个图或树，其中的节点代表可能的状态，边代表状态之间的转换。  选择搜索策略：在深度优先搜索中，你总是先访问一个节点的一个未访问过的子节点，直到没有未访问过的子节点为止，然后回溯到上一个节点。这种策略也被称为“先入后出”，因为最后一个被访问的节点总是最先被处理。  实施搜索：从起始节点开始，按照你选择的策略进行搜索。在搜索过程中，你需要记录哪些节点已经被访问过，以避免重复访问。  检查目标：每次访问一个新节点时，都检查它是否满足目标条件。如果满足，那么你就找到了一个解决方案。  回溯：如果当前节点不满足目标条件，或者所有子节点都已经被访问过，那么回溯到上一个节点，继续搜索。 |

# 宽度优先搜索

## 简介

### 原理

|  |
| --- |
| 宽度优先搜索的基本思想是从图的某个顶点开始，访问其所有相邻的顶点，然后对这些顶点的未访问过的邻居进行同样的操作，直到所有的顶点都被访问过。这种方法类似于水波从一个点开始向外扩散。 |

### 适用范围

|  |
| --- |
| 宽度优先搜索适用于许多问题，包括寻找图的连通分量、寻找最短路径（在无权图中）、解决迷宫问题等。此外，宽度优先搜索也常用于树的层次遍历。 |

### 步骤

|  |
| --- |
| 初始化：选择一个起始顶点，将其标记为已访问，并将其添加到一个“待访问”队列中。  搜索：从队列中取出一个顶点，访问该顶点的所有未访问过的邻居，将它们标记为已访问，并将它们添加到队列中。  结束：当队列为空时，算法结束。 |

## 案例

|  |
| --- |
| 树的创建 |

## 解题思路

|  |
| --- |
| 理解问题：首先，你需要理解问题的需求和限制条件。这将帮助你确定是否应该使用宽度优先搜索，以及如何定义搜索的目标和约束。  定义搜索空间：确定你需要搜索的空间。这通常是一个图或树，其中的节点代表可能的状态，边代表状态之间的转换。  选择搜索策略：在宽度优先搜索中，你总是先访问一个节点的所有未访问过的邻居，然后对这些邻居的未访问过的邻居进行同样的操作。这种策略也被称为“先入先出”，因为最先被访问的节点总是最先被处理。  实施搜索：从起始节点开始，按照你选择的策略进行搜索。在搜索过程中，你需要记录哪些节点已经被访问过，以避免重复访问。  检查目标：每次访问一个新节点时，都检查它是否满足目标条件。如果满足，那么你就找到了一个解决方案。 |

# 分治

## 简介

### 原理

|  |
| --- |
| 分治算法的基本思想是“分而治之”，即将一个大问题分解为几个相互独立的小问题，然后分别解决这些小问题，最后将这些小问题的解决方案合并为大问题的解决方案。这种方法的优点是可以将复杂的问题简化为更容易解决的小问题。 |

### 适用范围

|  |
| --- |
| 分治算法适用于许多问题，包括排序问题（如归并排序、快速排序）、搜索问题（如二分搜索）、数学问题（如幂运算、大整数乘法）等。分治算法特别适合于那些可以被分解为几个相互独立的小问题的问题。 |

### 步骤

|  |
| --- |
| 分解：将原问题分解为几个小问题。  解决：使用递归的方式解决这些小问题。如果小问题足够小，那么可以直接解决。  合并：将这些小问题的解决方案合并为原问题的解决方案。 |

## 案例

|  |
| --- |
| 理解问题：归并排序的目标是将一个无序的数组排序。  分解问题：我们将数组分解为两个较小的数组，每个数组包含原数组的一半元素。  解决子问题：我们递归地对这两个较小的数组进行归并排序。如果数组只包含一个元素，那么它已经是有序的，我们可以直接返回。  合并结果：我们将两个已排序的子数组合并为一个有序的数组。这就是原问题的解决方案。 |

## 解题思路

|  |
| --- |
| 理解问题：首先，你需要理解问题的需求和限制条件。这将帮助你确定是否应该使用分治算法，以及如何定义问题的目标和约束。  分解问题：将问题分解为几个较小的子问题。这些子问题应该是原问题的独立实例，可以独立解决，解决方法与原问题相同。  解决子问题：递归地解决每个子问题。如果子问题足够小，那么可以直接解决。  合并结果：将子问题的解决方案合并为原问题的解决方案。 |

# 匈牙利算法

## 案例

|  |
| --- |
| 给定一组数arr，从arr中挑出2个数，如相加为素数则结为伴侣。求最大伴侣数。 |

## 解题思路

|  |
| --- |
| 1、组成素数伴侣的必定是奇数+偶数；转换为二分图求最大匹配问题，左边是奇数，右边是偶数；  2、先统一求出左右两边可以匹配的组合，二维数组表示。  3、遍历左，依次寻找左边每个元素的增广路径。创建右的访问列表，每次find确保右每个元素不会重复访问  4、再遍历右，判断右的每个元素在没有被访问的情况下能否构成素数伴侣，满足则标记已访问。  5、右边元素是单身的情况下，直接构成素数伴侣，记录右边元素的伴侣，返回True  6、右边元素不是单身的情况，给右边元素的伴侣重新找一个。 |

# 双指针和滑动窗口

## 案例

|  |
| --- |
| 给定字符串，要求找出满足要求的最长子串的长度：   1. 子串任意字符最多出现2次 2. 子串不包含某个字符   如ABACA123D不包含A的最长子串长度为7 |

## 解题思路

|  |
| --- |
| 1. 定义双指针，l和r，都从0开始 2. 右指针+1，遇到新字符的数量大于2时，移动左指针，直到指针范围内小于2再继续 3. 遇到不允许的字符时，左右指针都跳过从右指针其后开始。 |

# 栈

## 案例

## 解题思路

|  |
| --- |
| 1. 符合条件的进栈 2. 触发结算时出栈，对出栈元素进行处理和判断，再入栈 |

# 二分查找

# 区间合并

# 并查集