0045. 跳跃游戏 II

ITCharge 大约 5 分钟

- 标签：贪心、数组、动态规划
- 难度：中等

题目链接

- 0045. 跳跃游戏 II - 力扣

题目大意

描述：给定一个非负整数数组 `nums`，数组中每个元素代表在该位置可以跳跃的最大长度。开始位置为数组的第一个下标处。

要求：计算出到达最后一个下标处的最少的跳跃次数。假设你总能到达数组的最后一个下标处。

说明：

- $1 \le nums.length \le 10^4$。
- $0 \le nums[i] \le 1000$。

示例：

- 示例 1：

```py
输入：nums = [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。
```

解题思路

思路 1：动态规划（超时）

1. 划分阶段

按照位置进行阶段划分。

2. 定义状态

定义状态 `dp[i]` 表示为：跳到下标 `i` 所需要的最小跳跃次数。

3. 状态转移方程

对于当前位置 `i`，如果之前的位置 `j` $(o \le j < i)$ 能够跳到位置 `i` 需要满足：位置 `j` $(o \le j < i)$ 加上位置 `j` 所能跳到的最远长度要大于等于 `i`，即 `j + nums[j] >= i`。

而跳到下标 `i` 所需要的最小跳跃次数等于满足上述要求的位置 `j` 中最小跳跃次数加 `1`，即 `dp[i] = min(dp[i], dp[j] + 1)`。

4. 初始条件

初始状态下，跳到下标 `0` 需要的最小跳跃次数为 `0`，即 `dp[0] = 0`。

5. 最终结果

根据我们之前定义的状态，`dp[i]` 表示为：跳到下标 `i` 所需要的最小跳跃次数。则最终结果为 `dp[size - 1]`。

思路 1：动态规划（超时）代码

```py
class Solution:
    def jump(self, nums: List[int]) -> int:
        size = len(nums)
        dp = [float("inf") for _ in range(size)]
        dp[0] = 0

        for i in range(1, size):
            for j in range(i):
                if j + nums[j] >= i:
                    dp[i] = min(dp[i], dp[j] + 1)

        return dp[size - 1]
```

思路 1：复杂度分析

- **时间复杂度**：$O(n^2)$。两重循环遍[illegible]时间复杂度是 $O(n^2)$，所以总体时间复杂度为 $O(n^2)$。
- **空间复杂度**：$O(n)$。用到了一维数组保存状态，所以总体空间复杂度为 $O(n)$。

思路 2：动态规划 + 贪心

因为本题的数据规模为 10^4，而思路 1 的时间复杂度是 $O(n^2)$，所以就超时了。那么我们有什么方法可以优化一下，减少一下时间复杂度吗？

上文提到，在满足 `j + nums[j] >= i` 的情况下，`dp[i] = min(dp[i], dp[j] + 1)`。

通过观察可以发现，`dp[i]` 是单调递增的，也就是说 `dp[i - 1] <= dp[i] <= dp[i + 1]`。

举个例子，比如跳到下标 `i` 最少需要 `5` 步，即 `dp[i] = 5`，那么必然不可能出现少于 `5` 步就能跳到下标 `i + 1` 的情况，跳到下标 `i + 1` 至少需要 `5` 步或者更多步。

既然 `dp[i]` 是单调递增的，那么在更新 `dp[i]` 时，我们找到最早可以跳到 `i` 的点 `j`，从该点更新 `dp[i]`。即找到满足 `j + nums[j] >= i` 的第一个 `j`，使得 `dp[i] = dp[j] + 1`。

而查找第一个 `j` 的过程可以通过使用一个指针变量 `j` 从前向后迭代查找。

最后，将最终结果 `dp[size - 1]` 返回即可。

思路 2：动态规划 + 贪心代码

```py
class Solution:
    def jump(self, nums: List[int]) -> int:
        size = len(nums)
        dp = [float("inf") for _ in range(size)]
        dp[0] = 0

        j = 0
        for i in range(1, size):
            while j + nums[j] < i:
                j += 1
            dp[i] = dp[j] + 1

        return dp[size - 1]
```

思路 2：复杂度分析

- **时间复杂度**：$O(n)$。最外层循环遍历的时间复杂度是 $O(n)$，看似和内层循环结合遍历的时间复杂度是 $O(n^2)$，实际上内层循环只遍历了一遍，与外层循环遍历次数是相加关系，两者的时间复杂度和是 $O(2n)$，$O(2n) = O(n)$，所以总体时间复杂度为 $O(n)$。
- **空间复杂度**：$O(n)$。用到了一维数组保存状态，所以总体空间复杂度为 $O(n)$。

思路 2：贪心算法

如果第 `i` 个位置所能跳到的位置为 `[i + 1, i + nums[i]]` ，则：

- 第 `0` 个位置所能跳到的位置就是 `[0 + 1, 0 + nums[0]]` ，即 `[1, nums[0]]` 。
- 第 `1` 个位置所能跳到的位置就是 `[1 + 1, 1 + nums[1]]` ，即 `[2, 1 + nums[1]]` 。
- ……

对于每一个位置 `i` 来说，所能跳到的所有位置都可以作为下一个起跳点，为了尽可能使用最少的跳跃次数，所以我们应该使得下一次起跳所能达到的位置尽可能的远。简单来说，就是每次在「可跳范围」内选择可以使下一次跳的更远的位置。这样才能获得最少跳跃次数。具体做法如下：

1. 维护几个变量：当前所能达到的最远位置 `end`，下一步所能跳到的最远位置 `max_pos`，最少跳跃次数 `setps`。
2. 遍历数组 `nums` 的前 `len(nums) - 1` 个元素：
3. 每次更新第 `i` 位置下一步所能跳到的最远位置 `max_pos`。
4. 如果索引 `i` 到达了 `end` 边界，则：更新 `end` 为新的当前位置 `max_pos`，并令步数 `setps` 加 `1`。
5. 最终返回跳跃次数 `steps`。

思路 2：贪心算法代码

```py
class Solution:
    def jump(self, nums: List[int]) -> int:
        end, max_pos = 0, 0
        steps = 0
        for i in range(len(nums) - 1):
            max_pos = max(max_pos, nums[i] + i)
            if i == end:
                end = max_pos
                steps += 1
        return steps
```

思路 2：复杂度分析

- **时间复杂度**：$O(n)$。一重循环遍历的时间复杂度是 $O(n)$，所以总体时间复杂度为 $O(n)$。
- **空间复杂度**：$O(1)$。只用到了常数项的变量，所以总体空间复杂度为 $O(1)$。

参考资料

- 【题解】【宫水三叶の相信科学系列】详解「DP + 贪心 + 双指针」解法，以及该如何猜 DP 的状态定义 - 跳跃游戏 II - 力扣
- 【题解】动态规划+贪心，易懂。 - 跳跃游戏 II - 力扣