# 基于 LCS 思路的最长严格递增子序列算法设计与分析

作者：郝星星 & ChatGPT

2024 年 12 月 23 日

## 1 问题背景与目标

在很多应用场景（例如数组处理、动态规划练习等）中，我们常常需要找出给定序列中最长的**严格递增子序列**（Longest Increasing Subsequence，简称 LIS）。严格递增子序列要求所选元素的下标严格递增，且所选元素的值也严格递增。例如，对于序列 $[3, 1, 5, 2, 6, 4]$，其所有严格递增子序列的一个示例是 $[3, 5, 6]$。本报告将参考 **LCS 问题**（Longest Common Subsequence，最长公共子序列）的思路，设计出一个用于寻找给定序列中 LIS 的算法，并在此基础上对算法进行进一步的优化。

## 2 LIS 的 $O(n^2)$ 算法

### 2.1 设计思路

参考 LCS 的动态规划思想：

1. 设有一个长度为 $n$ 的序列 $A = [a_1, a_2, \ldots, a_n]$。
2. 定义一个长度同为 $n$ 的一维数组 dp，其中 $\texttt{dp}[i]$ 表示**以 $a_i$ 结尾**的最长严格递增子序列的长度。
3. 初始时，每个位置都可以是长度为 1 的递增子序列，故 $\texttt{dp}[i] \leftarrow 1$。
4. 对于每个 $i$（从左到右），再向左扫描所有 $j < i$。若 $a_j < a_i$，则表示可以在以 $a_j$ 结尾的子序列后面接上 $a_i$ 形成更长的子序列，所以：
$$\texttt{dp}[i] = \max(\texttt{dp}[i], \texttt{dp}[j] + 1) \quad \text{前提是} a_j < a_i.$$
5. 最终答案即为 $\max\limits_{1 \le i \le n} \texttt{dp}[i]$。

该方法的时间复杂度主要由**两层循环**所决定，每次扫描需要 $O(n)$，共需要做 $n$ 次扫描，故整体时间复杂度为 $O(n^2)$。

### 2.2 伪代码示例

```
// 输入：长度为 n 的序列 A[1..n]
// 输出：A 的最长严格递增子序列的长度

function LIS_length_O_n2(A[1..n]):
    // Step 1: dp 数组初始化
    let dp[1..n] be an array
    for i from 1 to n:
```

```
    dp[i] <- 1

// Step 2: 动态规划填表
for i from 1 to n:
    for j from 1 to i-1:
        if A[j] < A[i]:
            dp[i] = max(dp[i], dp[j] + 1)

// Step 3: 答案为 dp 数组中的最大值
let ans = 0
for i from 1 to n:
    ans = max(ans, dp[i])
return ans
```

## 2.3 示例演示

我们以一个直观的例子来展示算法的执行过程。设有序列:

$$A = [5, 2, 8, 6, 3, 6, 9, 7].$$

依次计算 dp$[i]$ 的过程如下表所示:

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| $A[i]$ | 5 | 2 | 8 | 6 | 3 | 6 | 9 | 7 |
| $init\ dp[i]$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

| 更新过程 | |
|---|---|
| (1) i=1 | $dp[1]=1$ |
| (2) i=2 | $dp[2]=1$ |
| | 查看 j=1: $A[1]=5 \not< A[2]=2$ (不满足) |
| (3) i=3 | $dp[3]=1$ |
| | $j=1: A[1]=5<A[3]=8 \rightarrow dp[3]=\max(1, dp[1]+1)=2$ |
| | $j=2: A[2]=2<A[3]=8 \rightarrow dp[3]=\max(2, dp[2]+1)=2$ |
| (4) i=4 | $dp[4]=1$ |
| | $j=1: A[1]=5<A[4]=6 \rightarrow dp[4]=\max(1, dp[1]+1)=2$ |
| | $j=2: A[2]=2<A[4]=6 \rightarrow dp[4]=\max(2, dp[2]+1)=2$ |
| | $j=3: A[3]=8 \not< A[4]=6$(不满足) |
| (5) i=5 | $dp[5]=1$ |
| | $j=1: A[1]=5<A[5]=3$(不满足) |
| | $j=2: A[2]=2<A[5]=3 \rightarrow dp[5]=\max(1, dp[2]+1)=2$ |
| | $j=3: A[3]=8 \not< A[5]=3$(不满足) |
| | $j=4: A[4]=6 \not< A[5]=3$(不满足) |
| (6) i=6 | $dp[6]=1$ |
| | $j=1: A[1]=5<A[6]=6 \rightarrow dp[6]=\max(1, dp[1]+1)=2$ |
| | $j=2: A[2]=2<A[6]=6 \rightarrow dp[6]=\max(2, dp[2]+1)=2$ |
| | $j=3: A[3]=8 \not< A[6]=6$(不满足) |
| | $j=4: A[4]=6 \not< A[6]=6$(不满足, 严格递增需 $<$ ) |
| | $j=5: A[5]=3<A[6]=6 \rightarrow dp[6]=\max(2, dp[5]+1)=3$ |
| (7) i=7 | $dp[7]=1$ |
| | $j=1: A[1]=5<A[7]=9 \rightarrow dp[7]=\max(1, dp[1]+1)=2$ |
| | $j=2: A[2]=2<A[7]=9 \rightarrow dp[7]=\max(2, dp[2]+1)=2$ |
| | $j=3: A[3]=8<A[7]=9 \rightarrow dp[7]=\max(2, dp[3]+1)=3$ |
| | $j=4: A[4]=6<A[7]=9 \rightarrow dp[7]=\max(3, dp[4]+1)=3$ |
| | $j=5: A[5]=3<A[7]=9 \rightarrow dp[7]=\max(3, dp[5]+1)=3$ |
| | $j=6: A[6]=6<A[7]=9 \rightarrow dp[7]=\max(3, dp[6]+1)=4$ |
| (8) i=8 | $dp[8]=1$ |
| | $j=1: A[1]=5<A[8]=7 \rightarrow dp[8]=\max(1, dp[1]+1)=2$ |
| | $j=2: A[2]=2<A[8]=7 \rightarrow dp[8]=\max(2, dp[2]+1)=2$ |
| | $j=3: A[3]=8 \not< A[8]=7$(不满足) |
| | $j=4: A[4]=6<A[8]=7 \rightarrow dp[8]=\max(2, dp[4]+1)=3$ |
| | $j=5: A[5]=3<A[8]=7 \rightarrow dp[8]=\max(3, dp[5]+1)=3$ |
| | $j=6: A[6]=6<A[8]=7 \rightarrow dp[8]=\max(3, dp[6]+1)=4$ |
| | $j=7: A[7]=9 \not< A[8]=7$(不满足) |

| $final\ dp[i]$ | 1 | 1 | 2 | 2 | 2 | 3 | 4 | 4 |
|---|---|---|---|---|---|---|---|---|

由上表可见，$\max(dp)=4$。最长严格递增子序列的长度为 4。相应地，我们能找到若干个长度为 4 的递增子序列，其中一个是 [2, 3, 6, 9]，另一个是 [2, 3, 6, 7]（取最后一个元素 7 同样能形成长度 4）。

### 2.4 算法复杂度分析

从上述过程可知，该算法在最外层对每一个元素 $A[i]$ 进行遍历（$O(n)$），在内层对所有 $j < i$ 进行遍历（最多 $O(n)$），故总的时间复杂度为 $O(n^2)$。在多数通用场景下，该算法足以胜任规模中等（如 $n$ 在几千到几万）的应用。

# 3 $O(n \log n)$ 算法

如果想进一步提高效率，可借助以下思路将 LIS 问题在**平均或最坏情况下**实现到 $O(n \log n)$：

1. 准备一个辅助数组（有时称为“tail 数组”或者“牌顶数组”），用来记录当前长度的递增子序列的最后一个元素的最小可能值。
2. 从左到右遍历序列，每遇到一个新的元素 $a_i$，通过二分查找的方式在辅助数组中找到它应放置的位置，并进行更新。
3. 辅助数组的长度就是当前找到的**最长严格递增子序列长度**。

伪代码简单示例如下：

```
// 输入：长度为 n 的序列 A[1..n]
// 输出：LIS 的长度，时间复杂度 O(n log n)
function LIS_length_O_nlogn(A[1..n]):
    // tail[k] 表示“所有长度为 k 的递增子序列的末尾元素的最小值”
    create an empty array/list tail
    for i from 1 to n:
        // 在 tail 中寻找第一个 >= A[i] 的位置pos
        // 使用二分查找
        pos = binary_search_first_geq(tail, A[i])

        // 若 pos 等于 tail 长度，表示 A[i] 比 tail 中任何元素都大，直接插到末尾
        if pos == length(tail):
            tail.append(A[i])
        else:
            // 否则，将 tail[pos] 替换为 A[i]
            tail[pos] = A[i]

    return length(tail)
```

总体来说，**tail 数组的长度**始终保持“已能找到的最长递增子序列长度”，而对每个新元素 $a_i$ 的二分查找和替换操作只需 $O(\log n)$，遍历 $n$ 个元素后，总耗时 $O(n \log n)$。

# 4 总结

本报告首先通过一个 $O(n^2)$ 的**动态规划算法**演示了如何基于 LCS（最长公共子序列）问题的动态规划思路来解决最长严格递增子序列（LIS）问题，并在此基础上给出了一个 $O(n \log n)$ 的思路以便在数据规模更大时获得更高的效率。