华东师范大学数据科学与工程学院上机实践报告

课程名称：算法设计与分析　　**年级：**22 级　　**上机实践成绩：**

指导教师：金澈清　　**姓名：**郭夏辉

上机实践名称：最长上升子序列　　**学号：**10211900416　　**上机实践日期：**2023 年 5 月 11 日

上机实践编号：No.10　　**组号：**1-416

一、目的

1．熟悉算法设计的基本思想

2．掌握计算最长上升子序列的方法

二、内容与设计思想

1. 基于动态规划编写计算最长上升子序列方法的代码。
2. 基于贪心算法和二分搜索编写计算最长上升子序列方法的代码。
3. 对比两种实现方式不同数据量下运行时间的差异。

三、使用环境

推荐使用 C/C++集成编译环境。

四、实验过程

首先让我们再回顾一下什么是动态规划、我们利用动态规划算法的一般思路和对于最长上升子序列问题(LIS 问题，之后以此简写)我们应该怎么去分析和设计。

我们通常按如下 4 个步骤来设计一个动态规划算法：

1. **刻画一个最优解的结构特征。**
2. **递归地定义最优解的值。**
3. **计算最优解的值，通常采用自底向上的方法。**
4. **利用计算出的信息构造一个最优解。**

动态规划是一种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，在应用过程中需要严格地满足如下两个条件: 最优子结构(一个问题的最优解包含其子问题的最优解)和重叠子问题(问题的递归算法总是在反复地求解相同的子问题，而不是一直生成新的子问题)。

第一种方法称为带备忘的自顶向下法(top-down with memoization)[⊖]**。此方法仍按自然的递归形式编写过程，但过程会保存每个子问题的解(通常保存在一个数组或散列表中)。当需要一个子问题的解时，过程首先检查是否已经保存过此解。如果是，则直接返回保存的值，从而节省了计算时间；否则，按通常方式计算这个子问题。我们称这个递归过程是带备忘的**(memoized)**，因为它“记住”了之前已经计算出的结果。**

第二种方法称为自底向上法(bottom-up method)**。这种方法一般需要恰当定义子问题“规模”的概念，使得任何子问题的求解都只依赖于“更小的”子问题的求解。因而我们可以将子问题按规模排序，按由小至大的顺序进行求解。当求解某个子问题时，它所依赖的那些更小的子问题都已求解完毕，结果已经保存。每个子问题只需求解一次，当我们求解它(也是第一次遇到它)时，它的所有前提子问题都已求解完成。**

在回顾了相关的动态规划的基本知识之后，我们结合具体的 LIS 问题来看一下吧。

- LIS 问题

首先要明确子串和子序列的差异——两者虽然都是原始序列的子集，但是子串是连续的而子序列可以不保证连续性。LIS 问题要求我们在一个序列中找到一个子序列，它的数值严

格递增，并且使这个子序列的长度尽可能长。比如 A=[10,9,2,5,3,7,101,18]的最长递增子序列是[2,3,7,101]其长度是 4。

- 基于动态规划求解

为了求解 LIS 问题，我们最直观的思路是这样的，就是如果我们已知一个递增序列和其最后一个元素，向它追加一个值更大的元素，构成的新的子序列也是递增序列，这其实就满足了动态规划需要具有的最优子结构性质。在这个直观的解释之中，我们需要知道的是什么？一方面，是我们最关心的，递增序列有多长？另外一方面，也不可以忽视，就是最后一个元素是谁？(如果不知道这个，无从与可能要新加的元素来比较，那我们怎么敢轻易地追加呢？)

基于上面的分析，我们这样定义相关的状态量:dp[i]表示以 nums[i]结尾的最长上升子序列长度。对于边界情况，显然 dp[1]=1;对于一般情况，结合我们对可以追加的情况之论述，应该是这样子的:我们要求解 dp[i]，应该看的是 i 以前的 j 已经求出来的最优解,因此我们要在[1,i-1]范围内找到那个最优的 dp[j]。同时，为什么这个 i 位置要在 j 基础上能追加呢？这显然需要满足 num[i]>num[j]这个递增条件，只有这个也满足了，dp[i]才能在那个最优的 dp[j]之上再把自己加进来(+1)。我们很容易就可以列出相应的状态转移方程:

dp[i]=1 (如果 i 为 1) dp[i]=max{dp[j]}+1 (如果 num[i]>num[j],j 在[1,i-1]范围内迭代)在初始时，我们还要做一件事，就是给每个位置的 dp[i]赋值为 1（将自己作为一个序列）

假设 num 的长度是 n，这个方法的时间复杂度如何呢？在计算 dp[i]时，我们需要 O(n)的时间来遍历 dp[1]到 dp[i-1]的状态，所以总的时间复杂度是 O(n^2),还是比较大的。

- 基于贪心算法+二分搜索求解

为了优化 LIS 问题的求解，我们应该对相关的状态量之设置进行一些优化。结合朴素解法的状态转移方程 dp[i]=max{dp[j]}+1 (if num[i]>num[j] and j∈[1,i-1]),我们能发现精简化之后可以只有两个必要的状态量就能转移了:当前阶段最后一个元素和当前阶段序列的长度。我们从另外一个角度来分析，以当前阶段序列的长度而不是当前最后一个元素作为自变量，设 F[i]表示的是长度为 i 的 LIS 的末尾元素的最小值。之所以用的是最小值，这是因为一个形象化的分析——在长度给定的情况下，选择尽可能小的元素作为最末尾的值，之后的序列是以此为基础去拼接上来的，这样可以最大化选择出来的子序列之长度，进而满足我们的需求。与此同时，我们可以发现 F[]数组的值一定是单调不降的，这个性质使得我们在 F[]上进行二分查找成为了可能。那么，我们具体应该怎样去实现 LIS 呢？

我们从前到后去枚举原始数组 num[],每一次都对 F[]进行二分查找，找到那个小于 num[i]的最大 F[j],然后我们当然就可以去更新它的下一个 F[j+1]了啊，更新即可。利用这个方法，我们每次最多花费 O(lgn)的时间便可以更新状态，总的时间复杂度是 O(nlgn)，相比于朴素的 O(n^2)还是有很大程度的优化的。

- 基于树状数组的优化

这个方法并不是很常见，但是在实际应用过程中应该能有更好的优化效果，我在这里描述一下它。还是最朴素的方法的那个方程: dp[i]=max{dp[j]}+1 (if num[i]>num[j] and j∈[1,i-1]),结合基于贪心算法+二分搜索求解的讨论，我们只需要将比 i 小的所有的符合 num[j]<num[i]的 F[j]的最大值求出来，但是这个条件 num[j]<num[i]比较麻烦，我们再换一种思维方法：对于原序列每个元素，它有一个下标和一个权值，最长上升子序列实质就是求最多有多少元素它们的下标和权值都单调递增。

为何要用树状数组呢？因为树状数组是一个可以在指数级别时间复杂度下执行单点修改和区间查询操作的一种简洁的数据结构。这个问题的求解过程中涉及大量的有条件限定的在

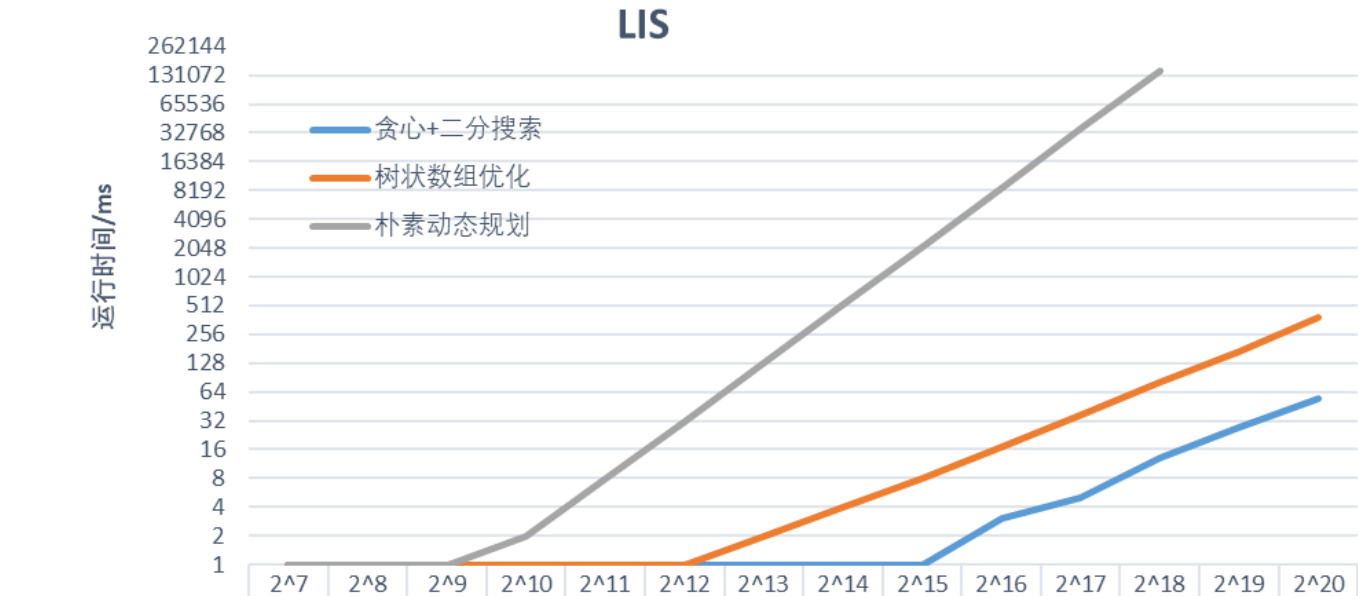
区间内搜索最大值之操作，可以利用树状数组将本来应该是 O(n)的时间复杂度优化为 O(lgn).

我们将 num 数组的每一元素先记住当前的下标，然后按照权值从小到大排序。然后按从小到大的顺序枚举 num 数组,这个时候值已经是单调递增了，对于状态的转移也就变成了从之前的标号比它小的状态转移过来。然后我们只需要建立一个以编号为下标同时维护长度的最大值的树状数组即可。枚举 num 数组时按元素的序号找到它之前序号比它小的长度最大的状态更新，然后将它也加入树状数组中。

利用树状数组优化，求解 LIS 的本质并没有变，只是说用一种更好的数据结构来优化其中的步骤。期望的时间复杂度还是 O(nlgn)。

五、总结

对上机实践结果进行分析，问题回答，上机的心得体会及改进意见。



	2^7	2^8	2^9	2^10	2^11	2^12	2^13	2^14	2^15	2^16	2^17	2^18	2^19	2^20
贪心+二分搜索	1	1	1	1	1	1	1	1	1	3	5	13	27	55
树状数组优化	1	1	1	1	1	1	2	4	8	17	37	80	170	380
朴素动态规划	1	1	1	2	8	31	128	519	2082	8794	36233	145800		

为了更加真实地对比各个方法求解 LIS 问题的时间差异，在给定数据规模时，我保持了三种方法的输入数据一致。前几次的实验我的图片有点粗糙，并没有很好地描述时间复杂度上的差异性——只看到了悬殊性，没看到清楚的对比，这次实验我采用了对数坐标轴的方法让结果更加直观。因为是对数化的纵轴，只能取正值，所以我把为小于 1ms 的时间按 1ms 计算了。

可以看到实验结果与最开始的分析基本是吻合的，随着数据规模的扩大，采用朴素的动态规划算法所消耗的时间明显增大，但是贪心+二分搜索方法和树状数组优化保持了 O(nlgn)这样一个可以接受的时间。有个比较有趣的现象是树状数组优化的时间复杂度前的常数更大，实际运用过程中还不如贪心+二分搜索方法。

通过本次实验，我不仅对 LIS 问题的求解有了一个更深刻的认识，而且还对动态规划算法和贪心算法的运用有了更高维度的掌握，希望自己之后能结合实际情况利用好它们，最优化程序的运行。