# 最长上升子序列

## 1　题目描述

广场上站着一支队伍，她们是来自全国各地的扭秧歌代表队，现在有她们的身高数据，请你帮忙找出身高依次递增的子序列。 例如队伍的身高数据是（1、7、3、5、9、4、8），其中依次递增的子序列有（1、7），（1、3、5、9），（1、3、4、8）等，其中最长的长度为4。

### 1.1　输入描述:

输入包含多组数据，每组数据第一行包含一个正整数n（1≤n≤1000）。紧接着第二行包含n个正整数m（1≤n≤10000），代表队伍中每位队员的身高。

### 1.2　输出描述:

对应每一组数据，输出最长递增子序列的长度。

### 1.3　输入例子:

7

1 7 3 5 9 4 8

6

1 3 5 2 4 6

### 1.4　输出例子:

4

4

## 2　解题思路

这题目是经典的DP题目，也可叫作LIS（Longest Increasing Subsequence）最长上升子序列或者最长不下降子序列。很基础的题目，有两种算法，复杂度分别为O()和O()。

### 2.1　O()算法分析

假设a[0:n]是给定的序列（长度为n+1，下标从0到n）

1. 对于a[n]来说，由于它是最后一个数，所以当从a[n]开始查找时，只存在长度为1的不下降子序列；
2. 若从a[n-1]开始查找，则存在下面的两种可能性：
3. 若a[n-1]<a[n]则存在长度为2的不下降子序列a[n-1]，a[n]；
4. 若a[n-1]>a[n]则存在长度为1的不下降子序列a[n-1]或者a[n]。
5. 一般若从a[t]开始，此时最长不下降子序列应该是按下列方法求出的：在a[t+1]，a[t+2]，...，a[n]中，找出一个比a[t]大的且最长的不下降子序列，作为它的后继。
6. 为算法上的需要，定义二个数组：

* 记录最大升序的数组len[0:n]：len[i]表示序列为a[0:i]，并且包含a[i]元素的最长子序列的长度。
* 记录后继位置的数组next[0:n]：next[i]表示i位置的下一个元素的位置是next[i]。

### 2.2　O()算法分析

设A[t]表示序列中的第t个数，F[t]表示从0到t这一段中以t结尾的最长上升子序列的长度，初始时设F[t]=0(t=1，2，...，len(A))。则有动态规划方程：

现在，我们仔细考虑计算F[t]时的情况。假设有两个元素A[x]和A[y]，满足

1. x<y<t
2. A[x]<A[y]<A[t]
3. F[x]=F[y]

此时，选择F[x]和选择F[y]都可以得到同样的F[t]值，那么，在最长上升子序列的这个位置中，应该选择A[x]还是应该选择A[y]呢？

很明显，选择A[x]比选择A[y]要好。因为由于条件(2)，在A[x+1]，...，A[t-1]这一段中，如果存在A[z]，A[x]<A[z]<A[y]，则与选择A[y]相比，将会得到更长的上升子序列。

再根据条件(3)，我们会得到一个启示：根据F[]的值进行分类。对于F[]的每一个取值k，我们只需要保留满足F[t]=k的所有A[t]中的最小值。设D[k]记录这个值，即D[k]=min{A[t]}(F[t]=k)。

注意到D[]的两个特点：

1. D[k]的值是在整个计算过程中是单调不下降的。
2. D[]的值是有序的，即D[0]<D[1]<D[2]<D[3]<...<D[n]。

利用D[]，我们可以得到另外一种计算最长上升子序列长度的方法。设当前已经求出的最长上升子序列长度为len。先判断A[t]与D[len]。若A[t]>D[len]，则将A[t]接在D[len]后将得到一个更长的上升子序列，len=len+1，D[len]=A[t]；否则，在D[0]...D[len]中，找到最大的j，满足D[j]<A[t]。令k=j+1，则有A[t]≤D[k]，将A[t]接在D[j]后将得到一个更长的上升子序列，更新D[k]=A[t]。最后，len即为所要求的最长上升子序列的长度。

在上述算法中，若使用朴素的顺序查找在D[0]...D[len]查找，由于共有O(n)个元素需要计算，每次计算时的复杂度是O(n)，则整个算法的 时间复杂度为O()，与原来的算法相比没有任何进步。但是由于D[]的特点(2)，我们在D[]中查找时，可以使用二分查找高效地完成，则整个算法 的时间复杂度下降为O(nlogn)，有了非常显著的提高。需要注意的是，D[]在算法结束后记录的并不是一个符合题意的最长上升子序列！

|  |
| --- |
| **private static int** lis2(**int**[] arr) {  **int**[] len = **new int**[arr.**length**];  **int**[] d = **new int**[arr.**length** + **1**];  // 使用最大值对d进行填充，保证在处理[0,k]时，单调递增Arrays.*fill*(d, Integer.***MAX\_VALUE***);  d[**0**] = -**1**; // 【1】   d[**1**] = arr[**0**]; // 【2】  len[**0**] = **1**; // 【3】  **for** (**int** i = **1**, j; i < arr.**length**; i++) { // 【4】  j = *find*(d, **0**, i, arr[i]); // 【5】  d[j] = arr[i]; // 【6】  len[i] = j; // 【7】  }  **int** max = **0**;  **for** (**int** i : len) { // 【8】  **if** (max < i) {  max = i;  }  }  **return** max; } |

对于这段程序，我们可以用算法导论上的loop invariants来帮助理解.

* loop invariant

1. 每次循环结束后d都是单调递增的。(这一性质决定了可以用二分查找）
2. 每次循环后，d[i]总是保存长度为i的递增子序列的最末的元素，若长度为i的递增子序列有多个，刚保存末尾元素最小的那个.（这一性质决定是第3条性质成立的前提）
3. 每次循环完后，b[i]总是保存以a[i]结尾的最长递增子序列。

* initialization:

1. 进入循环之前，d[0]=-1,d[1]=a[0],d的其他元素均为Integer.MAX\_VALUE,d是单调递增的;
2. 进入循环之前，d[1]=a[0],保存了长度为1时的递增序列的最末的元素，且此时长度为1的递增了序列只有一个，d[1]也是最小的;
3. 进入循环之前，b[0]=1，此时以a[0]结尾的最长递增子序列的长度为1.

* maintenance:

1. 若在第n次循环之前d是单调递增的，则第n次循环时，d的值只在第6行发生变化，而由d进入循环前单调递增及find函数的性质可知（见find的注释), 此时d[j+1]>d[j]>=a[i]>d[j-1],所以把d[j]的值更新为a[i]后，d[j+1]>d[j]> d[j-1]的性质仍然成立，即d仍然是单调递增的；
2. 循环中，d的值只在第6行发生变化，由d[j]>=a[i]可知，d[j]更新为a[i]后，d[j]的值只会变小不会变大，因为进入循环前d[j]的值是最小的，则循环中把d[j]更新为更小的a[i]，当然此时d[j]的值仍是最小的;
3. 循环中，b[i]的值在第7行发生了变化，因为有loop invariant的性质2，find函数返回值为j有：d[j-1]<a[i]<=d[j],这说明d[j-1]是小于a[i]的，且以d[j-1]结尾的递增子序列有最大的长度，即为j-1,把a[i]接在d[j-1]后可得到以a[i]结尾的最长递增子序列，长度为(j-1)+1=j;

* termination:

1. 循环完后，i=n-1,b[0],b[1],...,b[n-1]的值均已求出，即以a[0],a[1],...,a[n-1]结尾的最长递增子序列的长度均已求出，再通过第8行的循环，即求出了整个数组的最长递增子序列。

仔细分析上面的代码可以发现，每次循环结束后，假设已经求出d[1],d[2],d[3],...,d[len]的值，则此时最长递增子序列的长度为 len,因此可以把上面的代码更加简化，即可以不需要数组b来辅助存储，第8行的循环也可以省略。

|  |
| --- |
| **private static int** lis2(**int**[] arr) {  **int** len;  **int**[] d = **new int**[arr.**length** + **1**];  d[**0**] = -**1**;  d[**1**] = arr[**0**];  len = **1**;  **for** (**int** i = **1**, j; i < arr.**length**; i++) {  j = *find*(d, **0**, len, arr[i]);  d[j] = arr[i];  **if** (j > len) {  len = j;  }  }  // d[1:len]就是所求的上升序列**return** len; } |
| **private static int** find(**int**[] arr, **int** lo, **int** hi, **int** val) {  **int** mid;  **while** (lo <= hi) {  mid = lo + (hi - lo) / **2**;  **if** (arr[mid] < val) {  lo = mid + **1**;  } **else if** (arr[mid] > val){  hi = mid - **1**;  } **else** {  **return** mid;  }  }  **return** lo; } |