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($n\log n$)和O(n^2)。

2.1　O(n^2)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：

(1)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]；

(2)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]。

3、 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。

4、 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($n\log n$)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：

$$\mathbf{F}[t]=\boldsymbol{max}\{\mathbf{1}，\mathbf{F}[j]+\mathbf{1}\}(j=\mathbf{1}，\mathbf{2}，...，t-\mathbf{1}，\text{且}\mathbf{A}[j]<\mathbf{A}[t])$$

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 $O(n^2)$，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!

```java
    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。

```java
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;
        }
    }
```

```java
    // d[1:len]就是所求的上升序列
    return len;
}
```

```java
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;
}
```