3-2：

> *将习题3-1 中算法的计算时间减至$O(nlogn)$。（提示：一个长度为i 的候选子序列的最后一个元素至少与一个长度为$i-1$ 的候选子序列的最后一个元素一样大。通过指向输入序列中元素的指针来维持候选子序列）。*

题目分析

由于题干与 3-1 一致，因此此处不再分析题干，而是着眼于如何降低时间复杂度。

对于 3-1 复杂度为 $O(n^2)$ 的做法，原因在于遍历计算 DP 列表需 $O(n)$，而计算每个 $DP[k]$ 又需要 $O(n)$，两者嵌套导致 $O(n^2)$。其中，前者是无法降低的，而后者是可以降低的，接下来给出修改的方法：

我们需要用 $O(n)$的时间来计算每个 $DP[k]$，其问题就出在每次都需要从头开始更新 DP，若能通过某种方法，将更新 DP 的时间减至 $O(logn)$，那么一切便都迎刃而解。但我们发现，对于 DP 列表，必须自底向上更新，而每次又都必须从 $DP[0]$ 计算到 $DP[k]$，省略中间的任何一步都会导致错解。这个问题的出现于我们的状态设计不合理有关，我们的 DP 存的是长度，因此每取一个新数字 $L[p]$，都必须依次确认前面的“子序列的子序列”能否更优，于是我们的想法很自然就变为能不能存子序列末尾元素的值，而非子序列的长度？这样的话，由于越长的子序列其末尾值一定越大（或相等），因此，我们每次通过二分查找找到该 $L[p]$ 所能放入的位置，就可以实现 $O(logn)$ 复杂度了。

综上所述，需重新设计状态定义，我们现需要如下变量或列表：

- $Tail$：$Tail[i]$ 表示长度为 i 的子序列末尾元素的值。
- cur：cur 记录当前最长的单调递增子序列长度。
- pre，pos：回溯使用，将在下面详述。

于是我们遍历序列 L，遍历到 $L[k]$ 时，通过二分法遍历 $Tail[0, cur)$，找出 $L[k]$ 所处于的大小分界点：

- 如果在 $Tail[0]$ 到 $Tail[cur-1]$ 之中存在 p，使得 $Tail[p] > L[k]$，则将第一个满足的（也就是最小的 p）$Tail[p]$ 执行 $Tail[p] = L[k]$。
- 若不存在，说明 $L[k]$ 可以接在任何长度的子序列后，因此直接接到最长子序列的最后，并更新 cur，即 $Tail[++cur] = L[k]$。

从上面的思路可以发现，我们每次取 $L[k]$，都必然更新一个 $Tail[i]$ 的值，而且我们并不保证 $Tail[cur]$ 和其他 $Tail[cur-x] (x < cur)$ 有什么关系（详细说明见下方），因此，只要遍历完数组，我们所得到的 cur 便是最长单调递增子序列长度，而 $Tail[cur]$ 记录下来了一个最优的子序列的末尾元素值，同 3-1 一样，我们再维护一个 pre 数组，便可以构造出最优解。

而要构造最优解，我们不能再使用 3-1 的 pre 数组，因为我们底层的 $Tail$ 数组可能会发生变动，进而导致我们的不断回溯的指针造成混乱。因此我们还需要一个 pos 数组，$pos[i]$ 表示的是修改 $Tail[i]$ 时当时 $L[k]$ 的 k 值，譬如数字 $L[9] = 7$ 更新了 $Tail[3]$，则 $pos[3]$ = 9，$pre[i]$ 则表示第 i 个元素所代表的最长递增子序列前一个元素的下标，会与 $Tail$ 数组同步更新，由于 $Tail[k]$ 必从 $Tail[k-1]$ 拓展而来，因此只要用 $pos[k-1]$ 去更新 $pre[i]$，而 $pos[k-1]$ 存的就是当前最长递增子序列第 $k-1$ 个元素的下标，因此就能保证回溯的正确性。

** 对于 $L[k]$，要么它可以直接更新当前最长的递增子序列（替换末尾或直接接在后面）；要么则是对于长度为 $i(i < cur)$ 的递增子序列，可以直接替换末尾。在这个过程中，我们并未丢失最长子序列的信息，而同时我们会更新长度较小的递增子序列，这同样是一个“自底向上”的过程，若最终最优递增子序列为 S_k，则其必然由 $Tail[k-1]$ 更新而来，而 $Tail[k-1]$ 可以由 $Tail[k]$ 直接更新而来，也可以在 $Tail[k-1]$ 已经被更新后再更新而来（即 $Tail[k-1]$ 末尾元素的值经过一次更新变小，因此 $Tail[k]$ 就可能更优，最终就可以获得最优解。）

代码实现

详见 3-2 Code.cpp

输出示例

注：由于题干不变，所以示例与 3-1 一致。

```
输入1：
9
1 5 2 6 9 10 3 15 14
输出1：
长度为：6
1 2 6 9 10 14 // 最优子序列结尾应是14而非15

输入2：
4
4 3 2 1
输出2：
长度为：1
1

输入3：
15
3 5 4 7 3 2 1 6 3 4 6 9 8 11 12
输出3：
长度为：8
3 3 3 4 6 8 11 12

输入4：
10
3 -1 2 -4 6 3 5 7 -1 1
输出4：
长度为：5
-1 2 3 5 7

输入5：25
9 7 3 5 9 1 9 21 12 44 33 23 26 88 39 9 38 45 61 19 28 33 39 12 90
输出5：
长度为：11
3 5 9 9 12 23 26 28 33 39 90
```

算法分析

本算法与 3-1 的不同之处在于遍历 L 序列时，每次都使用 $upper\_bound$ 实现二分查找，因此每一次循环的复杂度均为 $O(logn)$，总复杂度为 $O(nlogn)$。