



程序设计实习

郭炜 微博 <http://weibo.com/guoweiofpku>

<http://blog.sina.com.cn/u/3266490431>

刘家瑛 微博 <http://weibo.com/pkuliujiaying>



北京大学
PEKING UNIVERSITY

信息科学技术学院《程序设计实习》 郭炜 刘家瑛

动态规划

最长上升子序列

例题二：最长上升子序列(百练2757)

问题描述

一个数的序列 a_i ，当 $a_1 < a_2 < \dots < a_s$ 的时候，我们称这个序列是上升的。对于给定的一个序列 (a_1, a_2, \dots, a_N) ，我们可以得到一些上升的子序列 $(a_{i_1}, a_{i_2}, \dots, a_{i_K})$ ，这里 $1 \leq i_1 < i_2 < \dots < i_K \leq N$ 。比如，对于序列 $(1, 7, 3, 5, 9, 4, 8)$ ，有它的一些上升子序列，如 $(1, 7)$ ， $(3, 4, 8)$ 等等。这些子序列中最长的长度是4，比如子序列 $(1, 3, 5, 8)$ 。

你的任务，就是对于给定的序列，求出最长上升子序列的长度。

输入数据

输入的第一行是序列的长度 N ($1 \leq N \leq 1000$)。第二行给出序列中的 N 个整数，这些整数的取值范围都在0到10000。

输出要求

最长上升子序列的长度。

输入样例

7

1 7 3 5 9 4 8

输出样例

4

解题思路

1. 找子问题

“求序列的前 n 个元素的最长上升子序列的长度”是个子问题，但这样分解子问题，不具有“无后效性”

假设 $F(n) = x$ ，但可能有多个序列满足 $F(n) = x$ 。有的序列的最后一个元素比 a_{n+1} 小，则加上 a_{n+1} 就能形成更长上升子序列；有的序列最后一个元素不比 a_{n+1} 小……以后的事情受如何达到状态 n 的影响，不符合“无后效性”

解题思路

1. 找子问题

“求以 a_k ($k=1, 2, 3\cdots N$) 为终点的最长上升子序列的长度”

一个上升子序列中最右边的那个数，称为该子序列的“终点”。

虽然这个子问题和原问题形式上并不完全一样，但是只要这 N 个子问题都解决了，那么这 N 个子问题的解中，最大的那个就是整个问题的解。

2. 确定状态：

子问题只和一个变量——数字的位置相关。因此序列中数的位置 k 就是“状态”，而状态 k 对应的“值”，就是以 a_k 做为“终点”的最长上升子序列的长度。

状态一共有 N 个。

3. 找出状态转移方程:

$\text{maxLen}(k)$ 表示以 a_k 做为“终点”的最长上升子序列的长度那么:

初始状态: $\text{maxLen}(1) = 1$

$$\text{maxLen}(k) = \max \{ \text{maxLen}(i) : 1 \leq i < k \text{ 且 } a_i < a_k \text{ 且 } k \neq 1 \} + 1$$

若找不到这样的 i , 则 $\text{maxLen}(k) = 1$

$\text{maxLen}(k)$ 的值, 就是在 a_k 左边, “终点”数值小于 a_k , 且长度最大的那个上升子序列的长度再加1。因为 a_k 左边任何“终点”小于 a_k 的子序列, 加上 a_k 后就能形成一个更长的上升子序列。

“人人为我” 递推型动归程序

```
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;
```

```
const int MAXN = 1010;
```

```
int a[MAXN]; int maxLen[MAXN];
```

```
int main() {
```

```
    int N;    cin >> N;
```

```
    for( int i = 1; i <= N; ++i) {
```

```
        cin >> a[i];
```

```
        maxLen[i] = 1;
```

```
    }
```

```
    for( int i = 2; i <= N; ++i) { //每次求以第i个数为终点的最长上升子序列的长度
```

```
        for( int j = 1; j < i; ++j) //察看以第j个数为终点的最长上升子序列
```

```
            if( a[i] > a[j] )
```

```
                maxLen[i] = max(maxLen[i], maxLen[j] + 1);
```

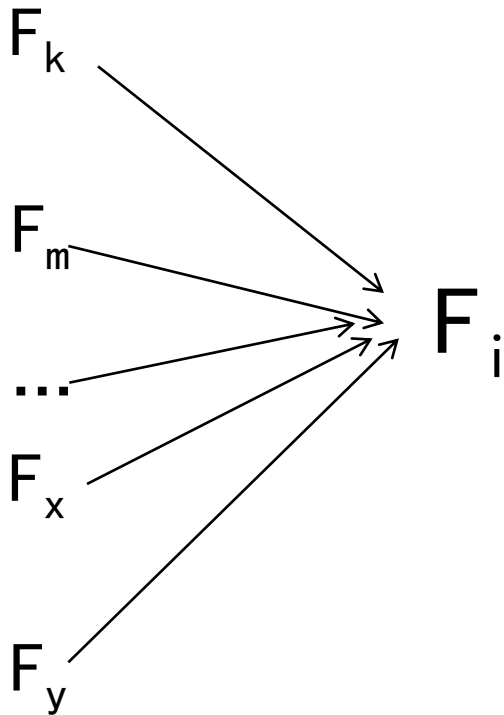
```
    }
```

```
    cout << * max_element(maxLen + 1, maxLen + N + 1 );
```

```
    return 0;
```

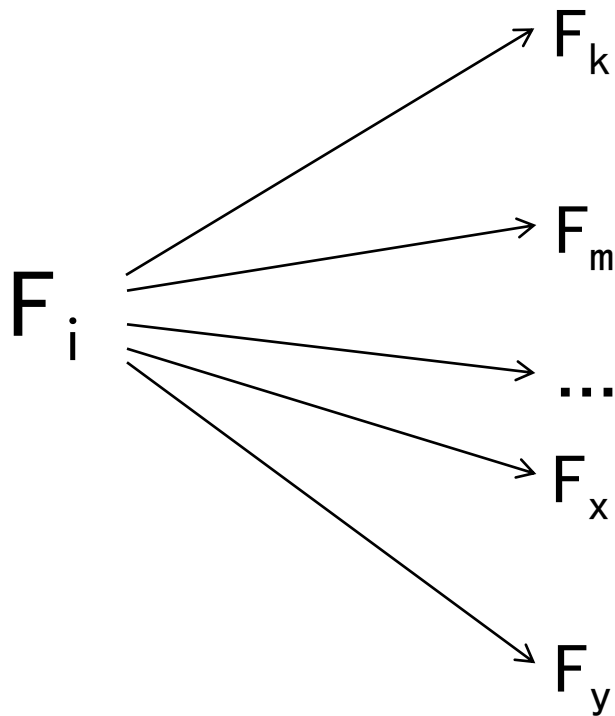
```
} //时间复杂度  $O(N^2)$ 
```

“人人为我” 递推型动归



状态 i 的值 F_i 由若干个值
已知的状态值 F_k, F_m, \dots, F_y
推出，如求和，取最大值
.....

“我为人人” 递推型动归



状态 i 的值 F_i 在被更新（不一定是最终求出）的时候，依据 F_i 去更新（不一定是最终求出）和状态 i 相关的其他一些状态的值
 F_k, F_m, \dots, F_y

“我为人人” 递推型动归程序

```
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;
```

```
const int MAXN = 1010;
```

```
int a[MAXN];
```

```
int maxLen[MAXN];
```

```
int main() {
```

```
    int N;    cin >> N;
```

```
    for( int i = 1; i <= N; ++i) {
```

```
        cin >> a[i];
```

```
        maxLen[i] = 1;
```

```
    }
```

```
    for( int i = 1; i <= N; ++i)
```

```
        for( int j = i + 1; j <= N; ++j ) //看看能更新哪些状态的值
```

```
            if( a[j] > a[i] )
```

```
                maxLen[j] = max(maxLen[j], maxLen[i] + 1);
```

```
    cout << * max_element(maxLen + 1, maxLen + N + 1 );
```

```
    return 0;
```

```
} //时间复杂度 $O(N^2)$ 
```

人人为我：

```
for( int i = 2; i <= N; ++i)
```

```
    for( int j = 1; j < i; ++j)
```

```
        if( a[i] > a[j] )
```

```
            maxLen[i] =
```

```
                max(maxLen[i], maxLen[j] + 1);
```

动归的三种形式

1) 记忆递归型

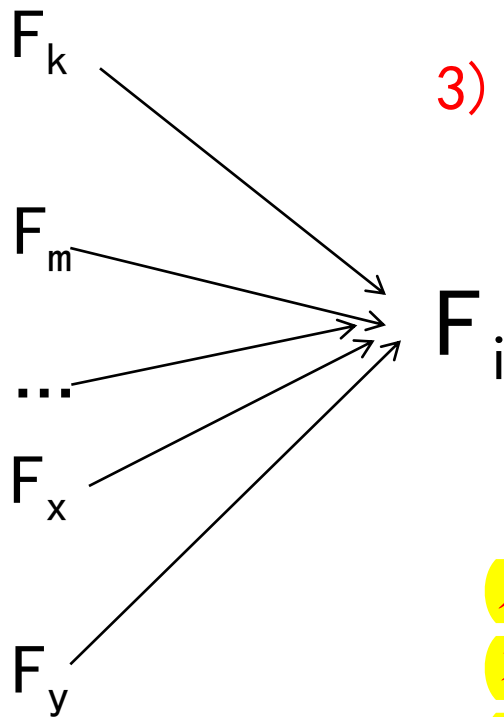
优点：只经过有用的状态，没有浪费。递推型会查看一些没用的状态，有浪费

缺点：可能会因递归层数太深导致爆栈，函数调用带来额外时间开销。总体来说，比递推型慢。

2) “我为人人”递推型

没有什么明显的优势，有时比较符合思考的习惯。个别特殊题目中会比“人人为我”型节省空间。

“人人为我”递推型中的优化



3) “人人为我”递推型

状态 i 的值 F_i 由若干个值
已知的状态值 F_k, F_m, \dots, F_y
推出，如求和，取最大值
.....

在选取最优备选状态的值 F_m, F_n, \dots, F_y 时，
有可能有好的算法或数据结构可以用来显
著降低时间复杂度。