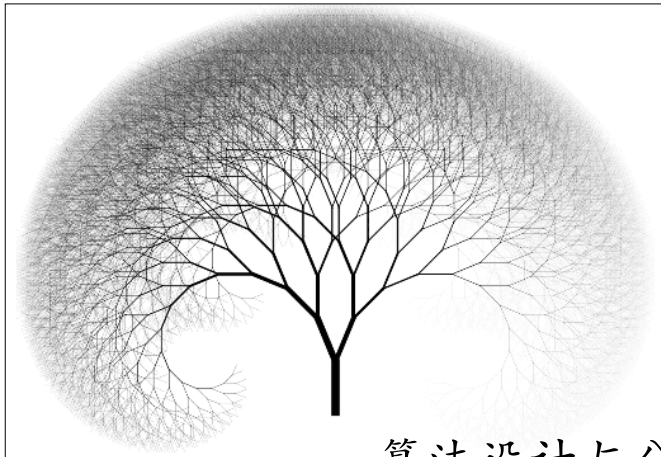


算法设计与分析

刘 安
苏州大学 计算机科学与技术学院
http://web.suda.edu.cn/anliu/

斐波那契数

Fibonnacci Numbers

斐波那契数

- $F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$ $\Rightarrow F(n) = \Theta(\phi^n)$ where $\phi = \frac{\sqrt{5}+1}{2} \approx 1.618$
- 递归算法

```
long fib_r(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib_r(n - 1) + fib_r(n - 2);
}
```

- $T(n)$：计算F_n时函数fib_r()的调用次数

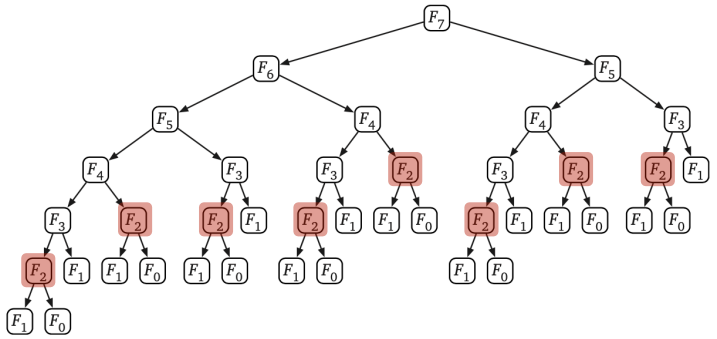
$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ T(n-1) + T(n-2) + 1 & \text{otherwise} \end{cases}$$

$\Rightarrow T(n) = 2F_{n+1} - 1$

$\Rightarrow T(n) = \Theta(\phi^n)$

fib_r()的递归调用树

- 以计算F_7为例



没有必要重复计算F_k的值
如果已经计算出F_k的值，不妨存储下来，下次需要的时候直接使用

自顶向下的动态规划：使用备忘录

- 使用备忘录，比如一个全局数组$f[0..n]$，来记录所有的F_k
  - 如果$f[k]$为-1，表示F_k值未知，需要计算；否则直接使用$F[k]$

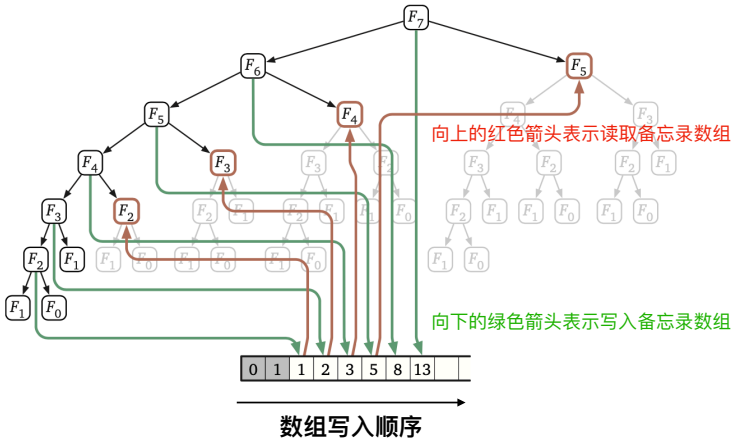
```
#define UNKNOWN -1
std::vector<long> f;

long fib_m(int n) {
    if (f[n] == UNKNOWN)
        f[n] = fib_m(n - 1) + fib_m(n - 2);
    return f[n];
}

long fib_m_driver(int n) {
    f = std::vector<long>(n + 1, UNKNOWN);
    f[0] = 0; f[1] = 1;
    return fib_m(n);
}
```

时间复杂度：$O(n)$

空间复杂度：$O(n)$

fib_m()的递归调用树

- 以计算F_7为例



自底向上的动态规划：主动填写备忘录

- 从左向右填写数组$f[0..n]$
  - 自左向右保证计算F_k的时候，F_{k-1}和F_{k-2}的值已经准备好

```
long fib_dp(int n) {
    f = std::vector<long>(n + 1, UNKNOWN);
    f[0] = 0; f[1] = 1;
    for (int i = 2; i <= n; i++)
        f[i] = f[i - 1] + f[i - 2];
    return f[n];
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

自底向上的动态规划：空间优化

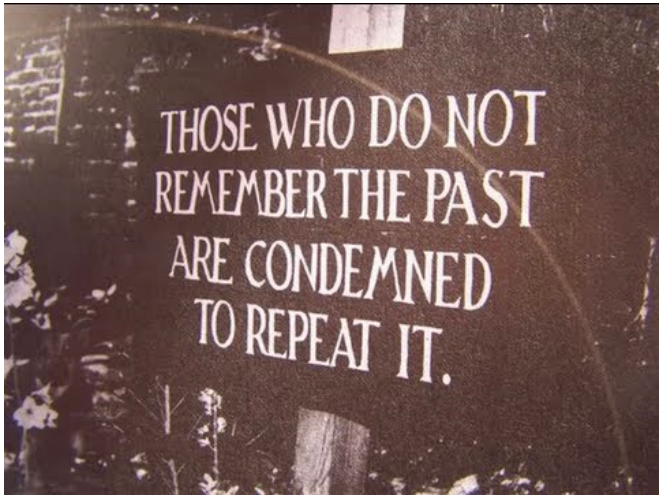
- $F_n = F_{n-1} + F_{n-2}$
  - 为了计算F_n，只需保存F_{n-1}和F_{n-2}的值，无需保存F_0至F_{n-3}的值

```
long fib_o(int n) {
    long prev = 1, curr = 0, next;
    for (int i = 1; i <= n; i++) {
        next = curr + prev;
        prev = curr;
        curr = next;
    }
    return curr;
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

动态规划小结

- 动机：在递归求解的过程中重复求解子问题
- 策略：通过空间换时间，将子问题的解存储下来，避免重复计算
  - 空间：子问题的数量
    - 子问题的数量由子问题的参数决定
      - 斐波那契数的子问题只有一个参数i且$0 \le i \le n$
- 关键：找到正确且高效的递归关系
- 求解方式
  - 自顶向下：被动填充备忘录，递归调用决定备忘录的填充顺序
  - 自底向上：主动填充备忘录，需要根据递归关系决定如何填充备忘录
    - 没有递归调用的时空开销



那些不能铭记历史的人注定要重蹈覆辙

— Dynamic Programming

思考

- Problem: given n, find the number of different ways to write n as the sum of 1, 3, 4
- Example: for $n = 5$, the answer is 6
  - 5 = 1 + 1 + 1 + 1 + 1
  - 5 = 1 + 1 + 3
  - 5 = 1 + 3 + 1
  - 5 = 3 + 1 + 1
  - 5 = 1 + 4
  - 5 = 4 + 1

二项式系数

Binomial Coefficients

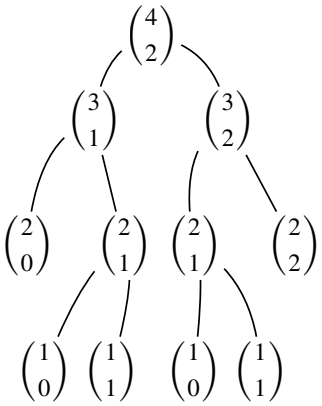
二项式系数

- $\binom{n}{k} = \frac{n!}{k!(n-k)!}$，$n \geq k \geq 0$
  - $\binom{20}{10} = 184756$，$20! = 2432902008176640000$，直接计算容易溢出
- 首先考虑能否递归求解
  - $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$
  - 基本情况

$$\binom{n}{k} \to \binom{n-1}{k-1} \to \cdots \to \binom{m}{0} = 1$$

$$\binom{n}{k} \to \binom{n-1}{k} \to \cdots \to \binom{k}{k} = 1$$

- 然后考虑是否重复求解子问题
  - 是，考虑使用动态规划



自顶向下的动态规划

- $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$，$\binom{n}{n} = 1$，$\binom{n}{0} = 1$

```
vector<vector<long>> bc;

long bc_m(int n, int k) {
    if (bc[n][k] == UNKNOWN)
        bc[n][k] = bc_m(n - 1, k - 1) + bc_m(n - 1, k);
    return bc[n][k];
}

long bc_m_driver(int n, int k) {
    bc = vector<vector<long>>(n + 1, vector<long>(n + 1, UNKNOWN));
    for (int i = 0; i <= n; i++) {
        bc[i][0] = 1;
        bc[i][i] = 1;
    }
    return bc_m(n, k);
}
```


自底向上的动态规划

- $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$，$\binom{n}{n} = 1$，$\binom{n}{0} = 1$
- 备忘录是一个二维数组
- 红色子问题的解依赖于两个黄色子问题的解
- 行优先填写备忘录
  - 自上向下逐行填写
  - 每一行从左向右或者从右向左
- 列优先填写备忘录
  - 从左向右逐列填写
  - 每一列自上向下
- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n^2)$

n/k	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1

自底向上的动态规划

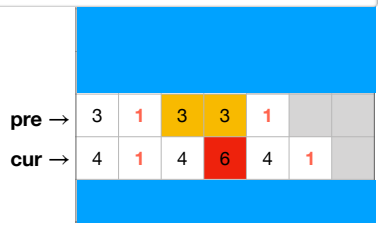
```
long bc_dp(int n, int k) {
    bc = vector<vector<long>>(n + 1, vector<long>(n + 1, UNKNOWN));
    for (int i = 0; i <= n; i++) {
        bc[i][0] = 1;
        bc[i][i] = 1;
    }
    for (int i = 2; i <= n; i++)
        for (int j = 1; j < i; j++)
            bc[i][j] = bc[i - 1][j - 1] + bc[i - 1][j];
    return bc[n][k];
}
```

n/k	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1

空间优化

```
long bc_dp_opt_space(int n, int k) {
    bc = vector<vector<long>>(2, vector<long>(n + 1, UNKNOWN));
    int cur = 0, pre = 1;
    for (int i = 0; i <= n; i++) {
        std::swap(cur, pre);
        bc[cur][0] = 1;
        bc[cur][i] = 1;
        for (int j = 1; j < i; j++)
            bc[cur][j] = bc[pre][j - 1] + bc[pre][j];
    }
    return bc[cur][k];
}
```

- 计算当前行的值只需要上一行的值
- 备忘录只需要保存两行
- 空间复杂度：$O(n)$



思考

- Problem: find a path from the upper-left corner to the lower-right corner of an $n \times n$ grid, with the restriction that we may only move down and right. Each square contains an integer, and the path should be constructed so that the sum of the values along the path is as large as possible.
- Example: the figure below shows an optimal path in a 5×5 grid. The sum of the values on the path is 67, and this is the largest possible sum on a path from the upper-left corner to the lower-right corner.

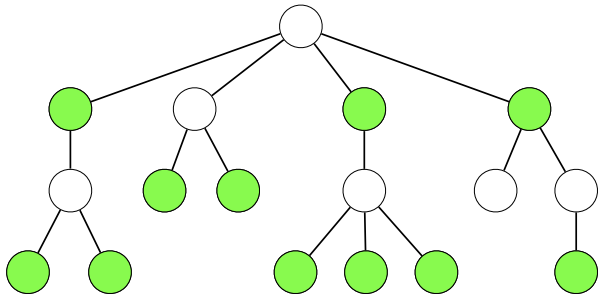
3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

树的最大独立集

Independent Set in a Tree

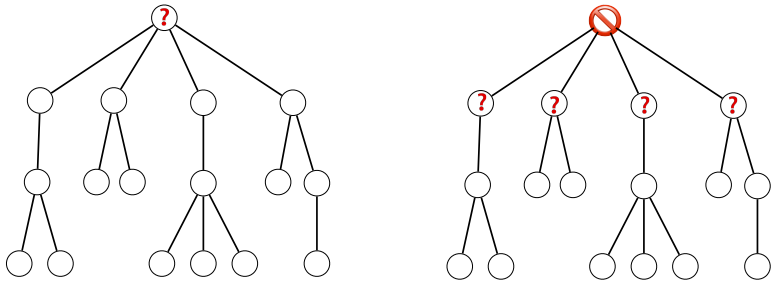
树的最大独立集

- 独立集：图中两两不相邻顶点构成的集合
- 最大独立集：不被任何其他独立集包含的独立集
- 问题：给定一棵树，计算它的最大独立集



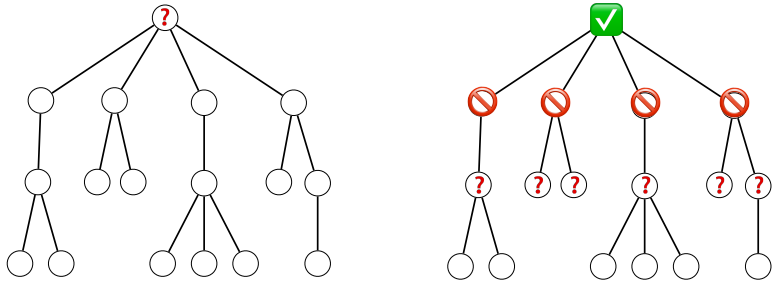
构造递归关系

- $MIS(v)$：以v为根的树的最大独立集IS的大小
  - v不在IS中
    - IS必然是以v的孩子为根的最大独立集的并集



构造递归关系

- $MIS(v)$：以v为根的树的最大独立集IS的大小
  - v在IS中
    - v的孩子肯定不在IS中
    - IS必然是以v的孩子的孩子为根的最大独立集的并集



设计动态规划算法

递归关系：$MIS(v) = \max\left\{\sum_{w\downarrow v} MIS(w), 1 + \sum_{w\downarrow v}\sum_{x\downarrow w} MIS(x)\right\}$

- 如何设计自底向上的动态规划算法
  - ☑构造递归关系
  - ☐备忘录形式
    - 通常情况下使用多维数组
    - 直接使用树来存储：每个节点v存储$MIS(v)$的值
  - ☐子问题的求解顺序
    - $MIS(v)$依赖于v的孩子以及孩子的孩子
    - 树的后序遍历：先遍历树的所有子树，再访问树的根节点

算法伪码

- 递归关系：$MIS(v) = \max\left\{\sum_{w\downarrow v} MIS(w),\ 1 + \sum_{w\downarrow v}\sum_{x\downarrow w} MIS(x)\right\}$
- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

```
tree_MIS(v)
    skip_v ← 0
    for each child w of v
        skip_v ← skip_v + tree_MIS(w)
    keep_v ← 1
    for each grandchild x of v
        keep_v ← keep_v + x.MIS
    v.MIS ← max{keep_v, skip_v}
    return v.MIS
```

另一种递归关系

- $MISyes(v)$：以v为根的树的包含v的最大独立集IS的大小
- $MISno(v)$：以v为根的树的不包含v的最大独立集IS的大小
- 原问题即求解$\max\{MISyes(r), MISno(r)\}$，其中$r$为树的根
- $MISyes(v) = 1 + \sum_{w \downarrow v} MISno(w)$
- $MISno(v) = \sum_{w \downarrow v} \max\{MISyes(w), MISno(w)\}$
- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

```
tree_MIS2(v)
    v.MISno ← 0
    v.MISyes ← 1
    for each child w of v
        v.MISno ← v.MISno + tree_MIS2(w)
        v.MISyes ← v.MISyes + w.MISno
    return max{v.MISyes, v.MISno}
```


最长递增子序列

Longest Increasing Subsequence

问题

- 子序列：对于任意序列s，它的子序列是通过删除其中零个或多个元素得到的另一个序列
  - 注意：剩余元素的相对顺序保持不变
- 给定n个整数组成的序列$s[1..n]$，求最长递增子序列LIS（的长度）

8	3	6	1	3	5	4	7

建立递归关系的一些思路

- 假设能够求出$s[1..k-1]$的LIS
  - 如果仅知道长度
    - 无法判断$s[k]$能否让LIS变长
  - 如果不仅知道长度，还知道具体序列L
    - 如果$s[k]$能让L变长，那么皆大欢喜
    - 否则
      - 也许L就是$s[1..k]$的LIS
      - 也许存在$s[1..k-1]$的另一个LIS：L'，$s[k]$能让L'变长
      - 可能需要记住$s[1..k-1]$的所有LIS

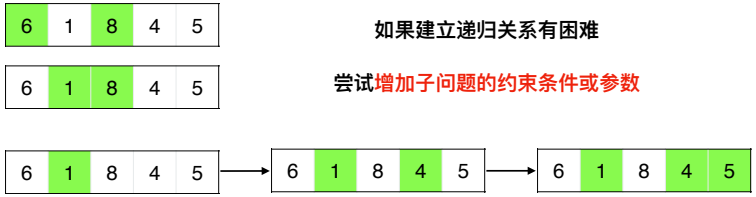
1	6		8

1	6	4	5

1	6	4	5

建立递归关系的一些思路

- 假设能够求出$s[1..k-1]$的所有LIS
  - $s[k]$不能让任一LIS变长
  - 但能让某个第二长的IS变长，并最终成为新的LIS
    - 所以还需要记住所有第二长的IS 😨
- 假设能够求出$s[1..k-1]$的所有最长和第二长的IS
  - 为了求第二长的IS，又需要知道所有第三长和第四长的IS 🤓

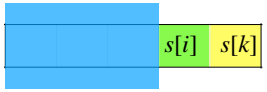


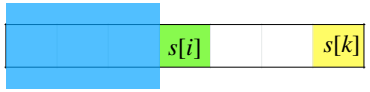
如果建立递归关系有困难

尝试增加子问题的约束条件或参数

增加子问题的约束条件

- 原始子问题：令$L(k)$表示$s[1..k]$的LIS的长度，原问题即求解$L(n)$
  - $O(n)$个子问题，但不容易建立递归关系
- 增加子问题的约束条件
  - 令$L(k)$表示$s[1..n]$中以$s[k]$结尾的LIS的长度
  - 原问题即为求解 $\max\limits_{1\le k\le n} L(k)$
  - 基本情况：如果$k=1$，那么$L(k)=1$
  - 归纳步骤
    - $L_k = \max\{1, \max\limits_{1\le i\le k-1}\{L(i)+1 \mid s[k]>s[i]\}\}$，其中，$\max\varnothing$的值定义为0





自底向上的动态规划

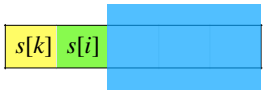
- 递归关系：$L(k)=\begin{cases}1 & \text{if } k=1\\ \max\{1, \max_{1\le i\le k-1}\{L(i)+1 \mid s[k]>s[i]\}\} & \text{if } k>1\end{cases}$
- $O(n)$个子问题
  - 计算$L(k)$的时间复杂度：$O(k)$
- 时间复杂度：$O(n^2)$

$s[1..n]$	8	3	6	1	3	5	4	7
$L(k)$	1	1	2	1	2	3	3	4

增加子问题的约束条件

- 如果令$L(k)$表示$s[1..n]$中以$s[k]$开头的LIS的长度
- 原问题？基本情况？归纳步骤？子问题数量？时间复杂度？

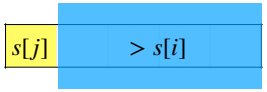
- 原问题即为求解 $\max\limits_{1\le k\le n} L(k)$
- 递归关系：$L(k)=\begin{cases}1 & \text{if } k=n\\ \max\{1, \max_{k+1\le i\le n}\{L(i)+1 \mid s[k]<s[i]\}\} & \text{if } k<n\end{cases}$
- $O(n)$个子问题
- 时间复杂度：$O(n^2)$

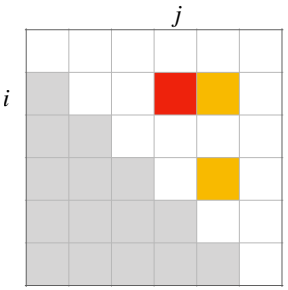


$s[1..n]$	8	3	6	1	3	5	4	7
$L(k)$	1	3	2	4	3	2	2	1

增加子问题的参数

- 额外的参数通常也引入新的约束条件
- 令$L(i,j)$表示$s[j..n]$中每个元素都大于$s[i]$的LIS的长度
- 令$s[0]=-\infty$，原问题即求解$L(0,1)$
- 基本情况：如果$j>n$，那么$L(i,j)=0$
- 归纳步骤
  - 如果$s[i]\geq s[j]$，$L(i,j)=L(i,j+1)$
  - 否则，$L(i,j)=\max\{L(i,j+1),1+L(j,j+1)\}$
- $O(n^2)$个子问题
- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n^2)$





自底向上的动态规划

$$\text{递归关系：}\ L(i,j)=\begin{cases}0 & \text{if } j>n\\ L(i,j+1) & \text{if } s[i]\geq s[j]\\ \max\begin{cases}L(i,j+1)\\1+L(j,j+1)\end{cases} & \text{otherwise}\end{cases}$$

$n=4$

$-\infty$	1	5	2	3
0	1	2	3	4

i \ j	0	1	2	3	4	5
0		3	2	2	1	0
1		2	2	2	1	0
2			0	0	0	0
3				1	1	0
4					0	0
5						0

$$OPT(0,4)=\max\begin{cases}OPT(0,5)\\1+OPT(4,5)\end{cases}=1$$

$$OPT(1,4)=\max\begin{cases}OPT(1,5)\\1+OPT(4,5)\end{cases}=1$$

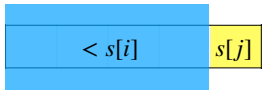
$$OPT(2,4)=OPT(2,5)=0$$

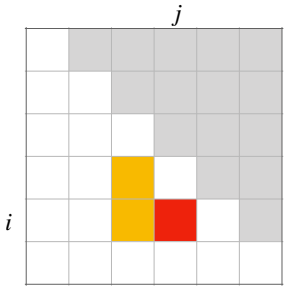
$$OPT(3,4)=\max\begin{cases}OPT(3,5)\\1+OPT(4,5)\end{cases}=1$$

$$OPT(4,4)=OPT(4,5)=0$$

增加子问题的参数

- 考虑后缀：令$L(i,j)$表示$s[j..n]$中每个元素都大于$s[i]$的LIS的长度
- 考虑前缀：令$L(i,j)$表示$s[1..j]$中**********的LIS的长度
  - 令$L(i,j)$表示$s[1..j]$中每个元素都小于$s[i]$的LIS的长度
- 令$s[n+1]=\infty$，原问题即求解$L(n+1,n)$
- 基本情况：如果$j=0$，那么$L(i,j)=0$
- 归纳步骤
  - 如果$s[i]\leq s[j]$，$L(i,j)=L(i,j-1)$
  - 否则，$L(i,j)=\max\{L(i,j-1),1+L(j,j-1)\}$
- $O(n^2)$个子问题：时空复杂度均为$O(n^2)$





自底向上的动态规划

$$\text{递归关系：}\ L(i,j)=\begin{cases}0 & \text{if } j=0\\ L(i,j-1) & \text{if } s[i]\leq s[j]\\ \max\begin{cases}L(i,j-1)\\1+L(j,j-1)\end{cases} & \text{otherwise}\end{cases}$$

$n=4$

1	5	2	3	∞
1	2	3	4	5

i \ j	0	1	2	3	4	5
0	0					
1	0	0				
2	0	1	1			
3	0	1	1	1		
4	0	1	1	2	2	
5	0	1	2	2	3	

$$OPT(1,1)=OPT(1,0)=0$$

$$OPT(2,1)=\max\begin{cases}OPT(2,0)\\1+OPT(1,0)\end{cases}=1$$

$$OPT(3,1)=\max\begin{cases}OPT(3,0)\\1+OPT(1,0)\end{cases}=1$$

$$OPT(4,1)=\max\begin{cases}OPT(4,0)\\1+OPT(1,0)\end{cases}=1$$

$$OPT(5,1)=\max\begin{cases}OPT(5,0)\\1+OPT(1,0)\end{cases}=1$$

能否继续优化

- 令$L(k)$表示$s[1..n]$中以$s[k]$结尾的LIS，原问题即为求解 $\max\limits_{1\le k\le n} L(k)$
- 递归关系：$L(k)=\begin{cases}1 & \text{if } k=1\\ \max\{1,\max_{1\le i\le k-1}\{L(i)+1\mid s[k]>s[i]\}\} & \text{if } k>1\end{cases}$
- 虽然只有$O(n)$个子问题，但时间复杂度是$O(n^2)$
  - 计算$L(k)$需要$O(n)$时间：遍历i来找最大的满足条件的$L(i)$：$s[i]>s[k]$
  - 能否更快？比如$O(\log n)$
    - $\Rightarrow O(n\log n)$
- 考虑计算$L[6]$
  - 长度为2：[3,6]和[1,3]
  - 长度为1：[8]、[3]和[1]

$s[1..n]$	8	3	6	1	3	5	4	7
$L(k)$	1	1	2	1	2	3	3	4

对于长度为k的IS，只需记住末尾元素最小的那个

$O(\log n)$时间计算$L(k)$

- 令$L(k)$表示$s[1..n]$中长度为k且末尾元素最小的递增子序列，且$L(k).last$表示该序列中最后那个元素
- 引理：$L(1).last<L(2).last<\cdots<L(k).last$
  - 假设$x\ge y$，而$y>z$，所以$x>z$
  - 那么灰色元素构成一个长度为k且末尾元素最小的递增子序列，矛盾

$L(k-1)$	…	…	x	
$L(k)$	…	…	z	y

- 归纳假设：对长度小于n的序列，可以计算其所有的$L(k)$，并有序存储
- 基本情况：长度为1的序列，那么$L[1]\leftarrow s[1]$
- 考虑如何基于归纳假设求解$s[1..n]$的所有的$L(k)$
  - 在$L(k).last$构成的有序数组中查找插入位置k'，使得$s[n]$加入后仍然有序
  - 如果$k'=k+1$，那么$L(k+1)\leftarrow L(k)+1$且$L(k+1).last\leftarrow s[n]$
  - 否则$L(k').last\leftarrow s[n]$，但$L(k')$的值不变
  - 时间复杂度：$O(\log n)$

升序

L(1).last	L(2).last	…	L(k).last

$s[n]$

运行实例

8	3	6	1	3	5	4	7

	1	2	3	4	5	6	7	8
8								
3								
6								
1								
3								
5								
4								
7								