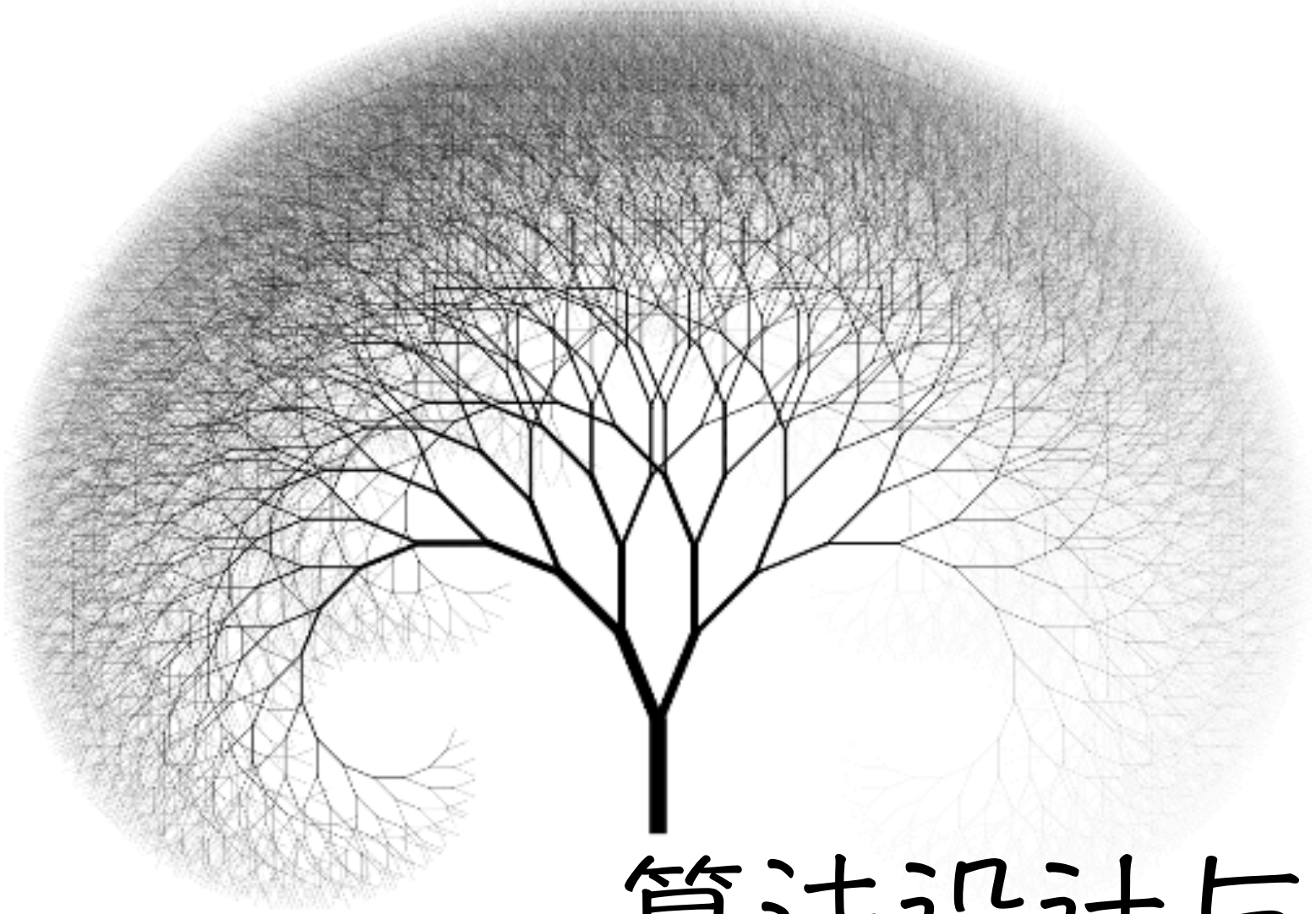
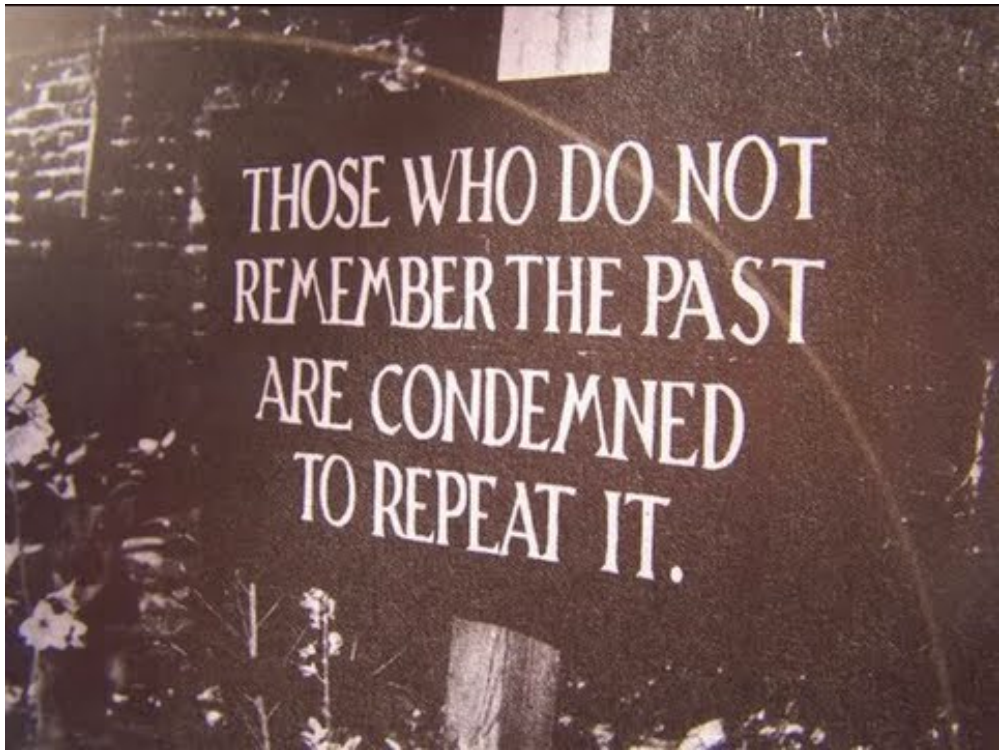


算法设计与分析

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

动态规划

- 斐波那契数列
- 二项式系数
- 加权任务调度
- 文本分割
- 最长递增子序列
- 编辑距离
- 背包问题
- 最优二叉搜索树
- 树的最大独立集
- 动态规划的局限性



那些不能铭记过去的人注定要重蹈覆辙

斐波那契数列

- 定义

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

- 递归算法

```
def rec_fibo(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return rec_fibo(n-1) + rec_fibo(n-2)
```

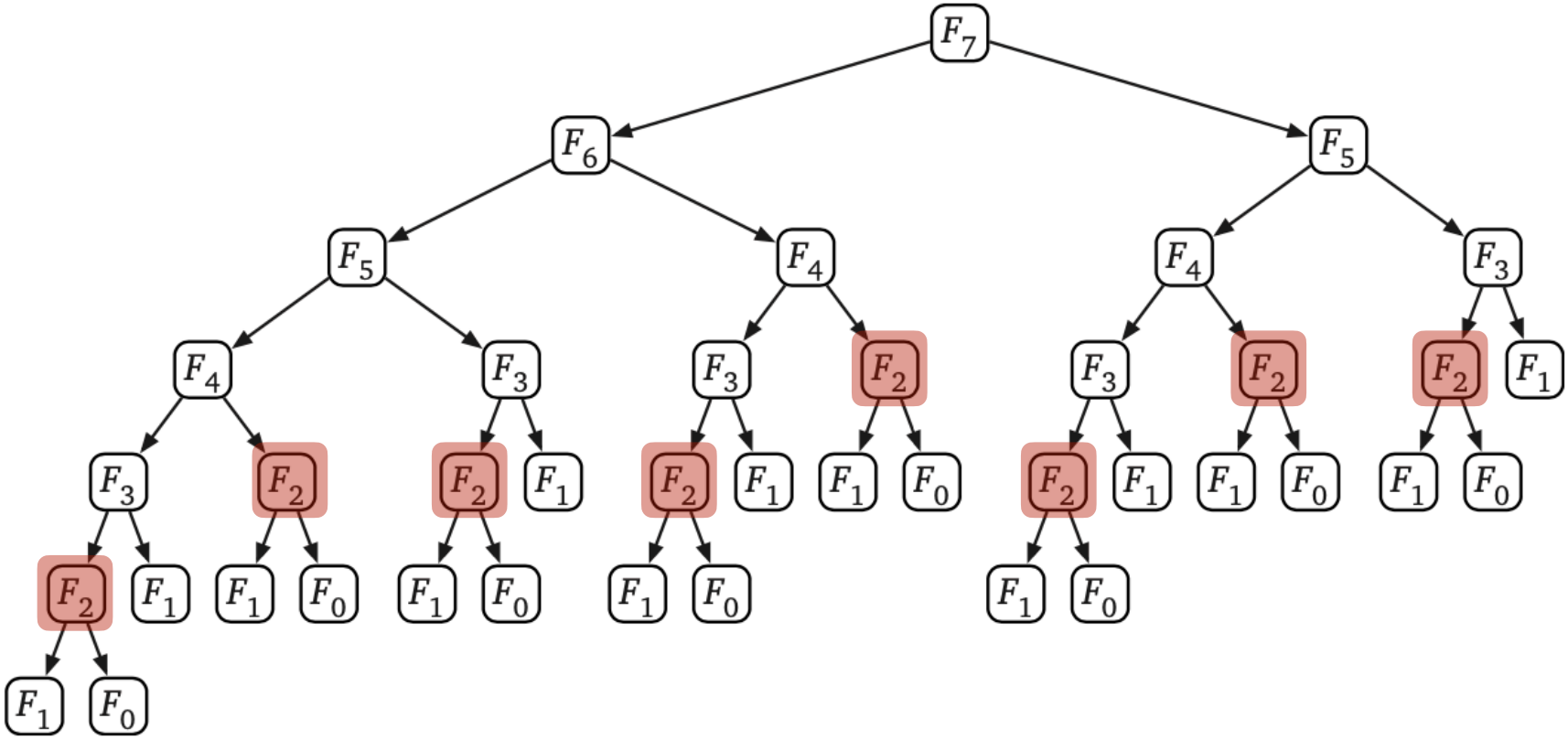
- $T(n)$：计算F_n时函数rec_fibo()的调用次数

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

n	0	1	2	3	4	5	6	7	8
F	0	1	1	2	3	5	8	13	21
T	1	1	3	5	9	15	25	41	67

rec_fibo()的递归调用树

- 以计算F_7为例



没有必要重复计算F_2的值

如果已经计算出F_2的值，不妨存储下来，下次需要的时候直接使用就好

自顶向下的动态规划：使用备忘录

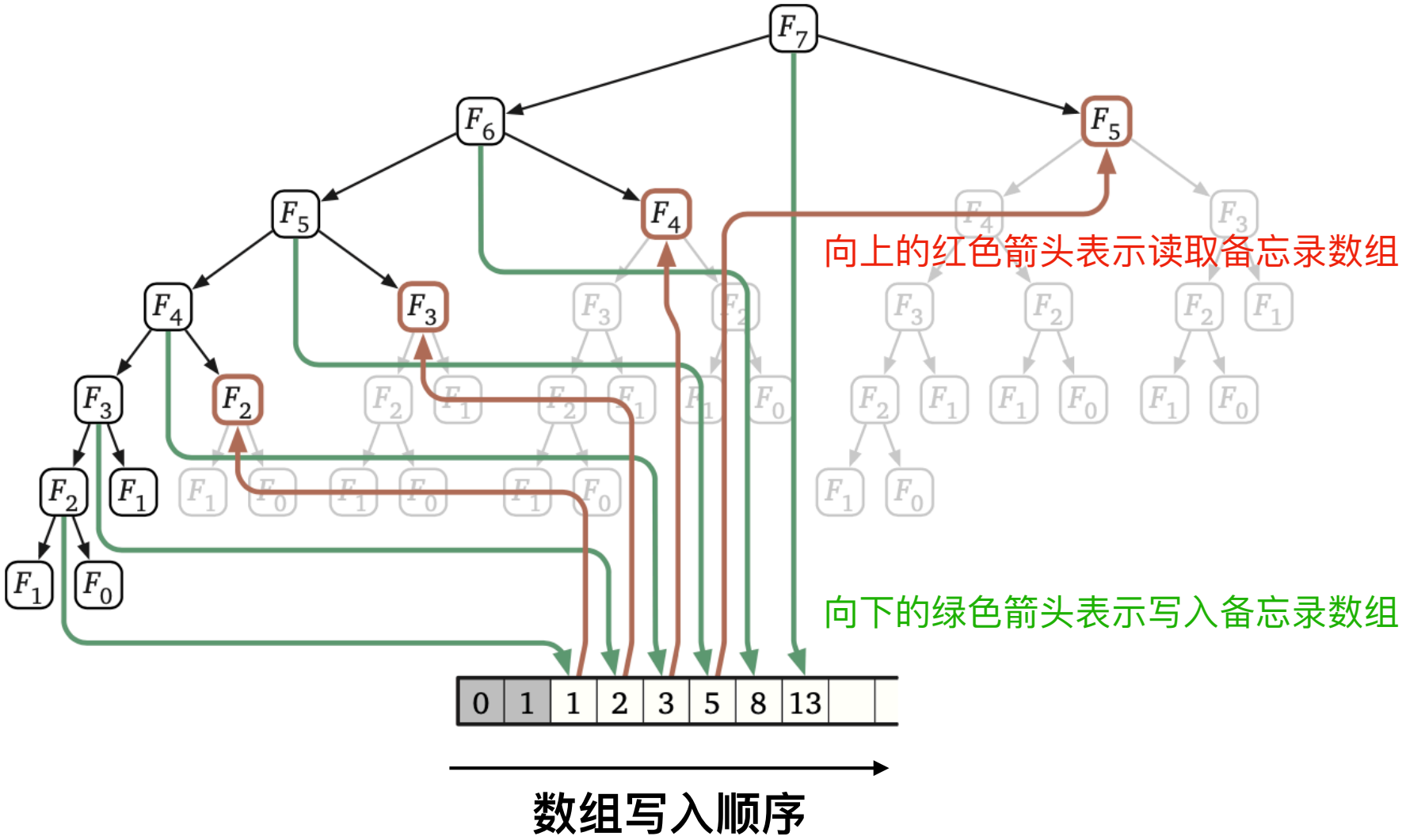
- 使用备忘录，即一个全局数组$F[0..n]$，来记录所有中间结果
  - 如果$F[k]$为$None$，表示需要计算F_k，否则直接使用$F[k]$

```
F = [None] * (n + 1)
F[0] = 0
F[1] = 1
def memo_fibo(n):
    if F[n] == None:
            F[n] = memo_fibo(n-1) + memo_fibo(n-2)
    return F[n]
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

mem_fibo()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

- 迭代算法
  - 从左向右填写数组$F[0..n]$

```
def iter_fibo(n):
    F[0] = 0
    F[1] = 1
    for i in range(2, n + 1):
        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

- $F_n = F_{n-1} + F_{n-2}$
  - 没有必要记录所有中间结果，只需要前2项即可

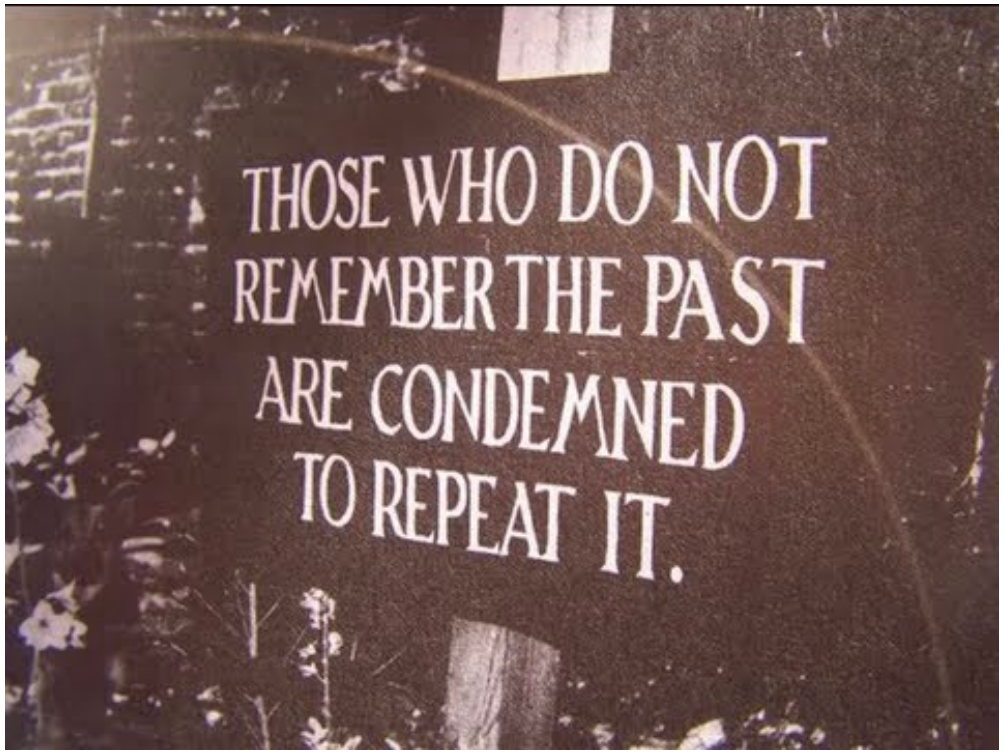
```python
def iter_fibo_2(n):
    prev = 1
    curr = 0
    for i in range(n):
        next = curr + prev
        prev = curr
        curr = next
    return curr
```

i		0	1	2	3	4
prev	1	0	1	1	2	3
curr	0	1	1	2	3	5
next		1	1	2	3	5

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

动态规划

- 斐波那契数列
- 二项式系数
- 加权任务调度
- 文本分割
- 最长递增子序列
- 编辑距离
- 背包问题
- 最优二叉搜索树
- 树的最大独立集
- 动态规划的局限性



那些不能铭记过去的人注定要重蹈覆辙

二项式系数

- $\binom{n}{k} = \frac{n!}{k!(n-k)!}, \quad n \geq k \geq 0$
  - $\binom{20}{10} = 184756, \quad 20! = 2432902008176640000,$ 直接计算容易溢出
- $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$

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

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

- 计算$\binom{20}{10}$，函数调用次数为369511

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

```
def n_choose_k(n, k):
    if n == k:
        return 1
    elif k == 0:
        return 1
    else:
        return n_choose_k(n-1, k-1) + n_choose_k(n-1, k)
```

二项式系数

- $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
  - $\binom{20}{10} = 184756, \quad 20! = 2432902008176640000$
- $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$

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

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

- 子问题的求解顺序
- 根据递归式确定子问题的依赖关系

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

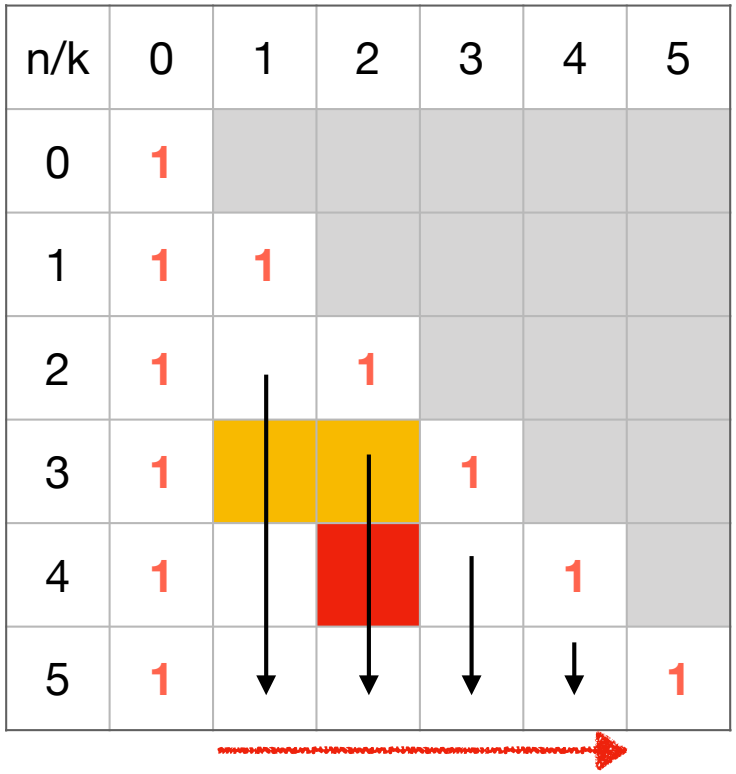
二项式系数

- $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
  - $\binom{20}{10} = 184756, \quad 20! = 2432902008176640000$
- $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$

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

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

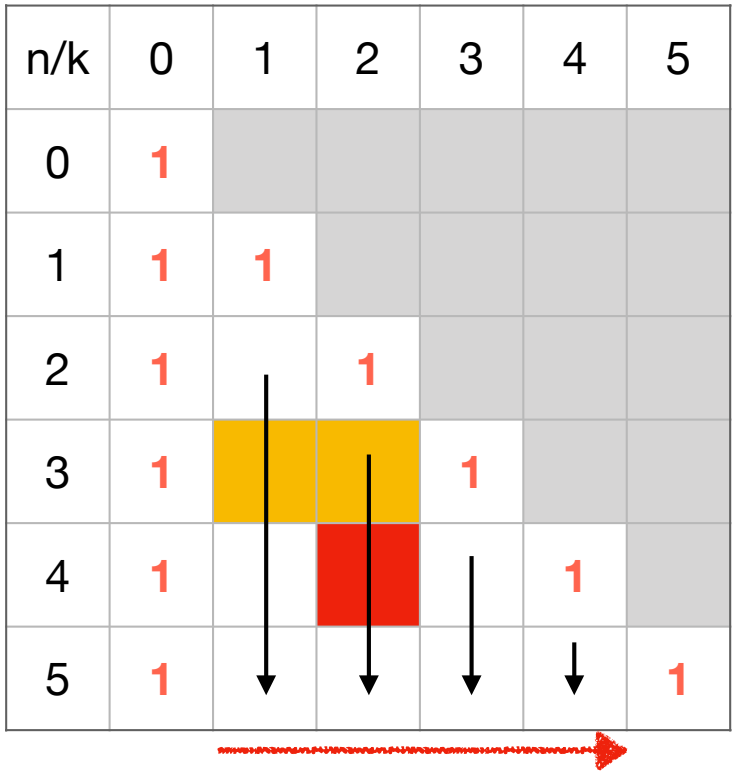
- 列优先填写备忘录
  - 从左向右逐列填写
  - 每一列自上而下



算法实现

- 列优先填写备忘录
  - 从左向右逐列填写
  - 每一列自上而下

```
def n_choose_k_iter(n, k):
    M = [[None] * (k+1) for _ in range(n+1)]
    for i in range(n + 1):
        M[i][0] = 1
    for j in range(k + 1):
        M[j][j] = 1
    for j in range(1, k + 1):
        for i in range(j + 1, n + 1):
            M[i][j] = M[i-1][j-1] + M[i-1][j]
    return M[n][k]
```



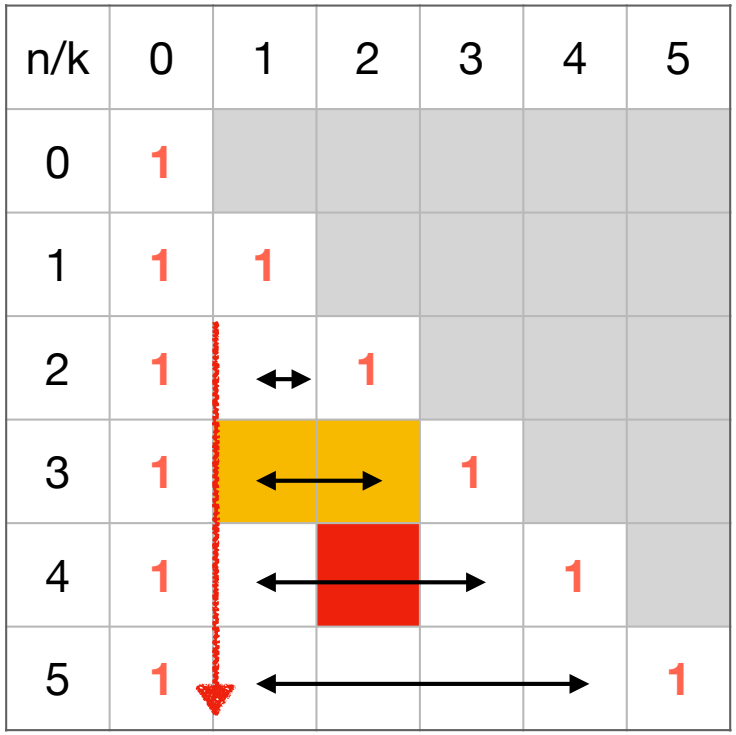
二项式系数

- $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
  - $\binom{20}{10} = 184756, \quad 20! = 2432902008176640000$
- $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$

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

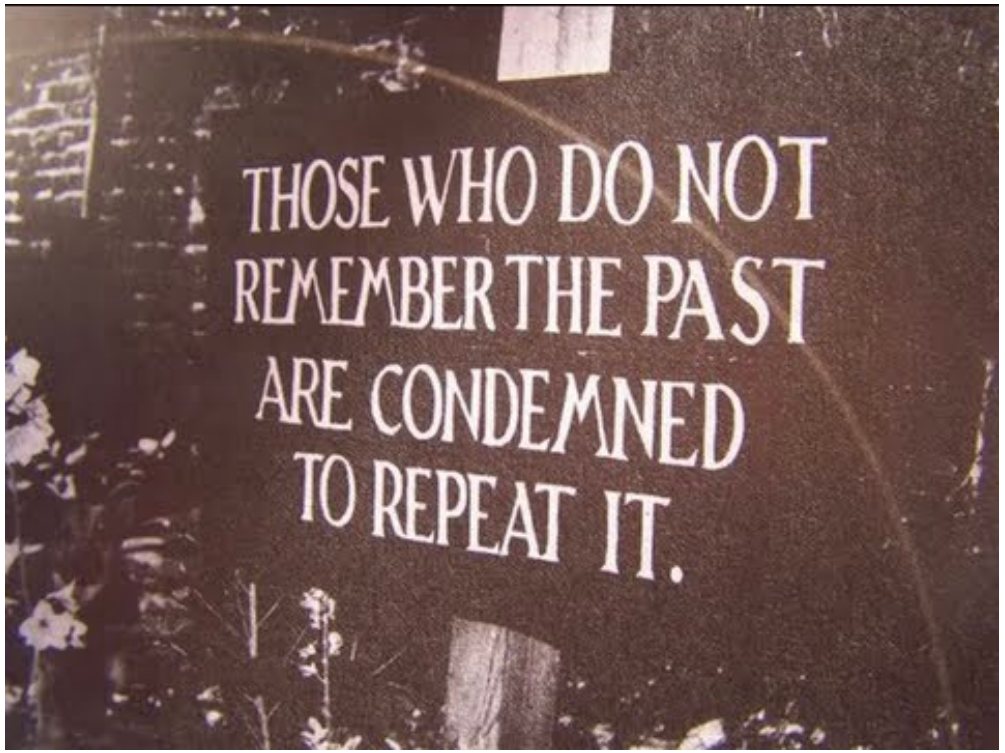
$$\binom{n}{k} \to \binom{n-1}{k} \to \cdots \to \binom{m}{m} = 1$$

- 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



动态规划

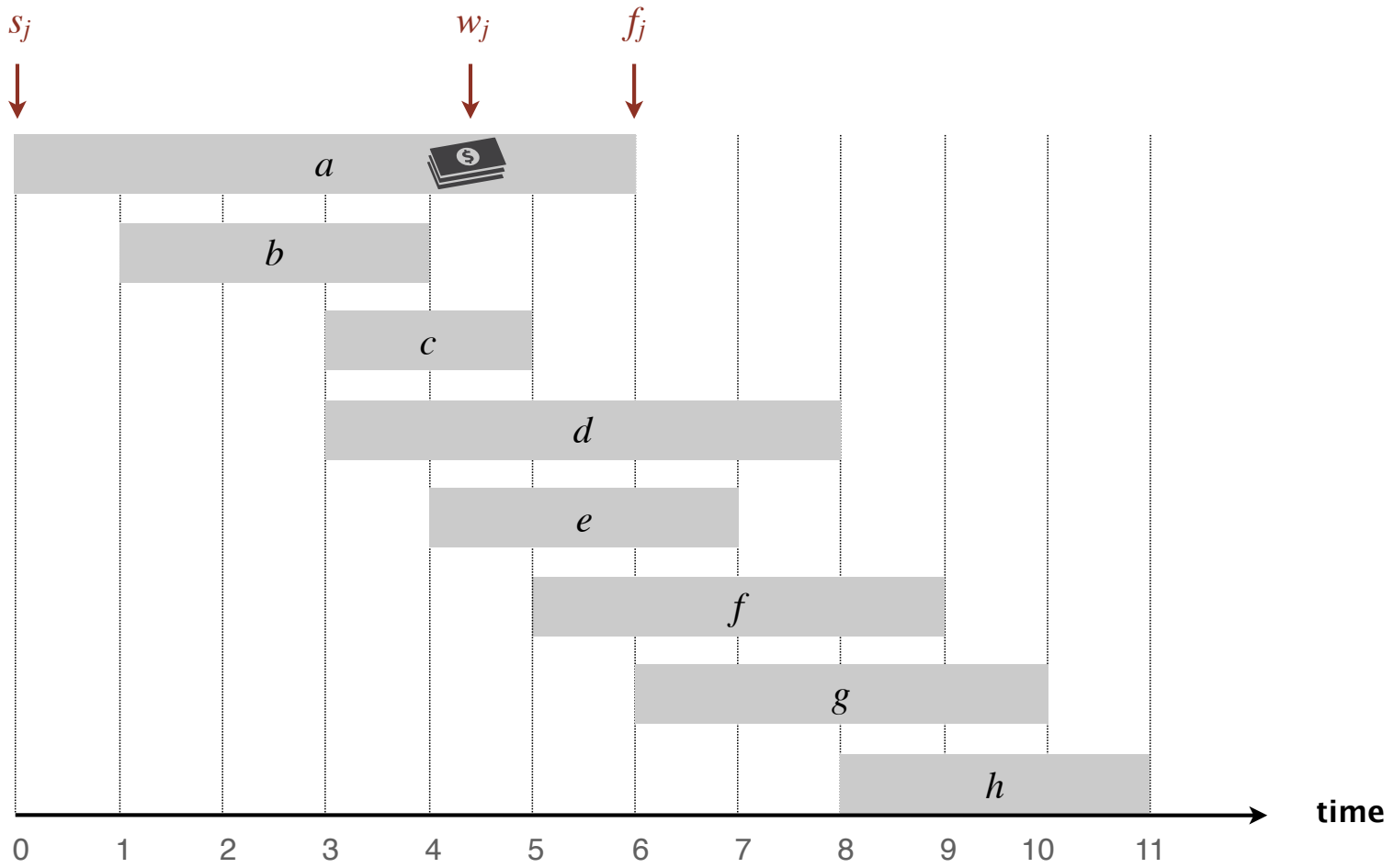
- 斐波那契数列
- 二项式系数
- 加权任务调度
- 文本分割
- 最长递增子序列
- 编辑距离
- 背包问题
- 最优二叉搜索树
- 树的最大独立集
- 动态规划的局限性



那些不能铭记过去的人注定要重蹈覆辙

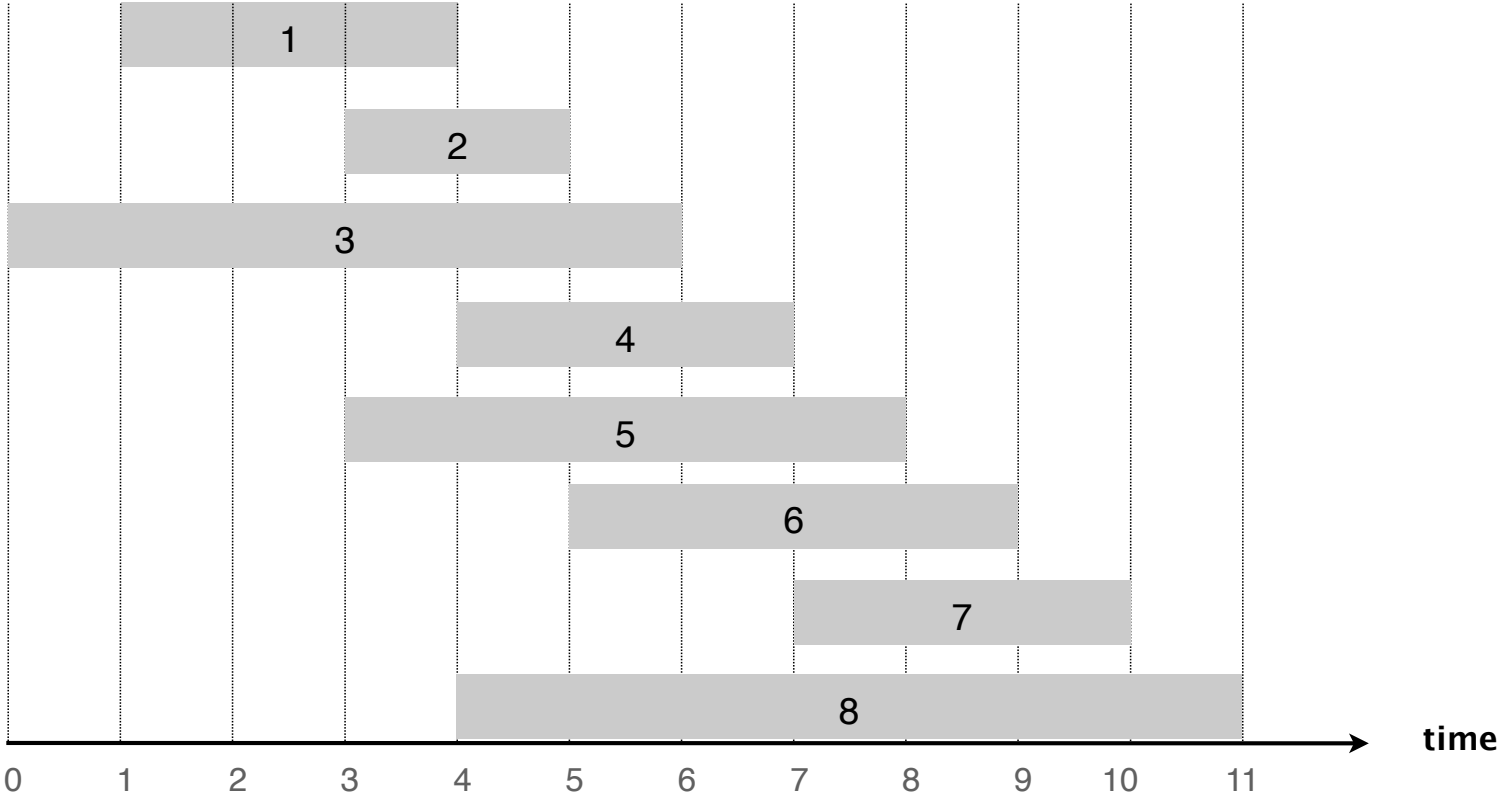
加权任务调度

- 已知：任务j的开始时间s_j，结束时间f_j，权重$w_j > 0$
- 如果两个任务在执行时间上没有重叠，它们是不相交的
- 问题：找到一个具有最大权重的互不相交的任务子集



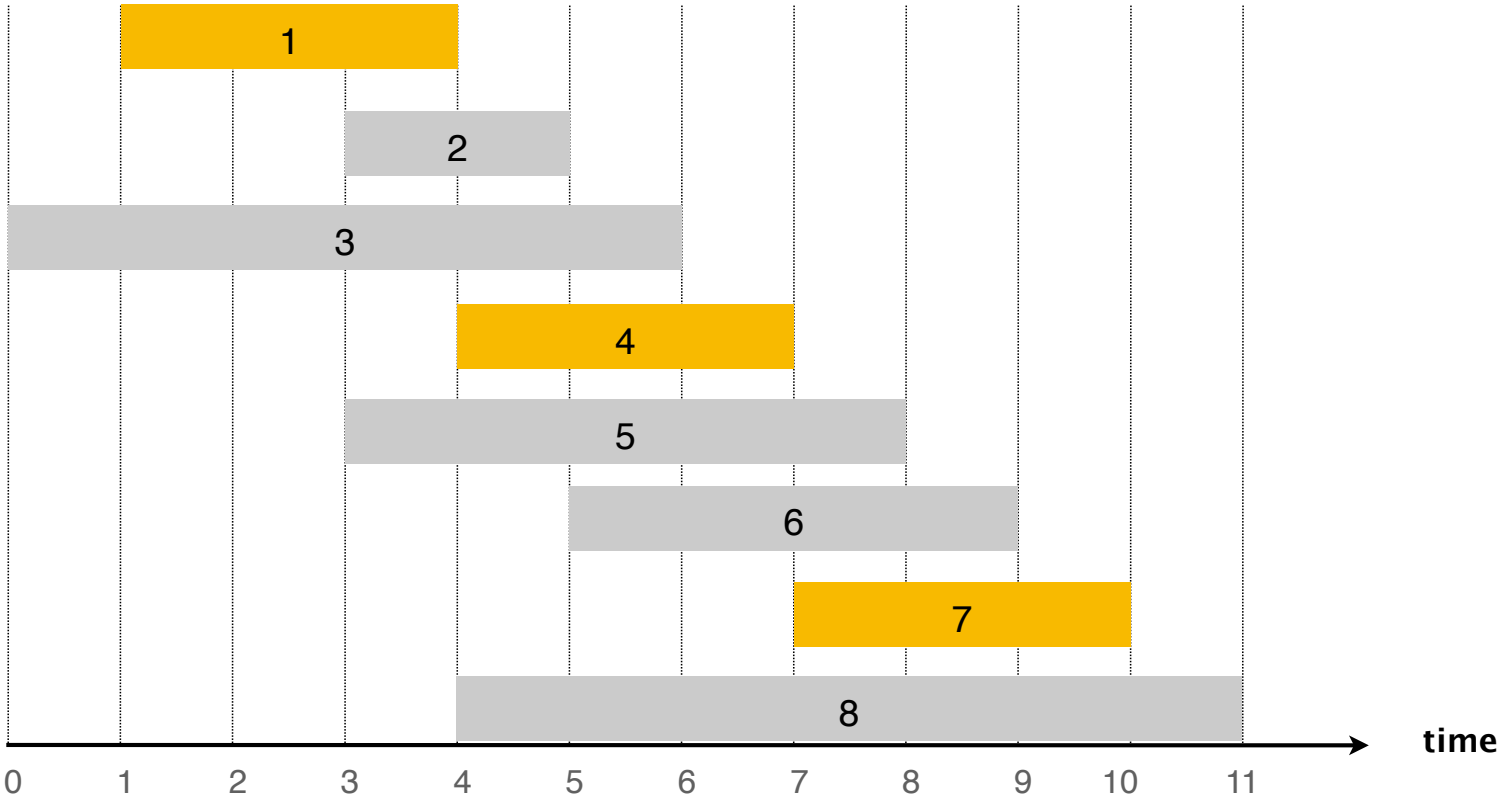
加权任务调度

- 假设所有任务已经按照它们的结束时间升序排列，即$f_1 \le f_2 \le \cdots \le f_n$
- 考虑在任务j开始之前就已结束的任务集合$\{k | f_k \le s_j\}$，其中i是最晚结束的任务，即$i = \max\{k | f_k \le s_j\}$，令$p(j) = i$
  - $p(8) = 1, p(7) = 4, p(2) = 0$
  - 在任务j之前完成且与任务j不相交的任务是：$1,2,\cdots,p(j)$



使用二元选择构造递归

- 给定任务$1,2,\cdots,n$，令最优解是O_n，其值（即权重）是$OPT(n)$
- 对于O_n，下面两种情况之一必然成立
  - O_n不包含任务n：O_n必然是子问题（包括任务$1,2,\cdots,n-1$）的最优解
  - O_n包含任务n：O_n必然是子问题（包括任务$1,2,\cdots,p(n)$）的最优解



使用二元选择构造递归

- 给定任务$1,2,\cdots,n$，令最优解是O_n，其值（即权重）是$OPT(n)$
- 对于O_n，下面两种情况之一必然成立
  - O_n不包含任务n　　$OPT(n) = OPT(n-1)$
    - O_n必然是子问题（包括任务$1,2,\cdots,n-1$）的最优解
  - O_n包含任务n　　$OPT(n) = w_n + OPT(p(n))$
    - 任务n贡献了权重w_n
    - O_n必然是子问题（包括任务$1,2,\cdots,p(n)$）的最优解
- 递归式

$$OPT(n) = \begin{cases} 0 & \text{if } n = 0 \\ \max\left\{ OPT(n-1), w_n + OPT(p(n)) \right\} & \text{if } n > 0 \end{cases}$$

问题的最优解由子问题的最优解组成

自底向上的动态规划

- 全局数组$M[0..n]$存储所有子问题最优解的值
- 时间复杂度：$O(n \log n)$
  - 如何计算$p[j]$
    - 二分搜索

Algorithm 1: `WeightedJobScheduling`$(n, s_1, \cdots, s_n, f_1, \cdots, f_n, w_1, \cdots, w_n)$

```
1 sort jobs by finish time and renumber so that f_1 ≤ f_2 ≤ ··· ≤ f_n
2 compute p[1], p[2], ··· , p[n]
3 M[0] ← 0
4 for j ← 1 to n do
5     M[j] ← max{M[j − 1], w_j + M[p[j]]}
```

算法运行实例

任务	1	2	3	4	5	6	7	8
开始时间	1	2	0	4	3	5	6	4
结束时间	4	5	6	7	8	9	10	11
权重	1	1	1	1	1	1	1	1
p[j]	0	0	0	1	0	2	4	1

Algorithm 1: `WeightedJobScheduling`$(n, s_1, \cdots, s_n, f_1, \cdots, f_n, w_1, \cdots, w_n)$

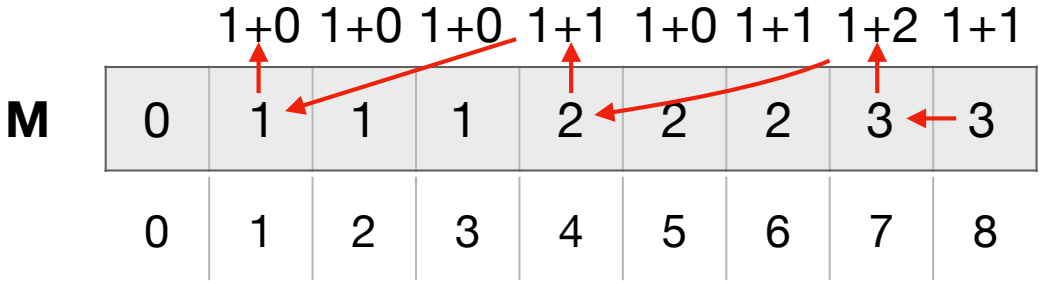
```
sort jobs by finish time and renumber so that f_1 ≤ f_2 ≤ ··· ≤ f_n
compute p[1], p[2], ··· , p[n]
M[0] ← 0
for j ← 1 to n do
    M[j] ← max{M[j − 1], w_j + M[p[j]]}
```

		1+0	1+0	1+0	1+1	1+0	1+1	1+2	1+1
M	0	1	1	1	2	2	2	3	3
	0	1	2	3	4	5	6	7	8

构造最优解

任务	1	2	3	4	5	6	7	8
开始时间	1	2	0	4	3	5	6	4
结束时间	4	5	6	7	8	9	10	11
权重	1	1	1	1	1	1	1	1
p[j]	0	0	0	**1**	0	2	**4**	1

- $OPT(8) = \max\{OPT(7), w_8 + OPT(1)\}$
- $OPT(7) = \max\{OPT(6), w_7 + OPT(4)\}$
- $OPT(4) = \max\{OPT(3), w_4 + OPT(1)\}$
- $OPT(1) = \max\{OPT(0), w_1 + OPT(0)\}$



构造最优解

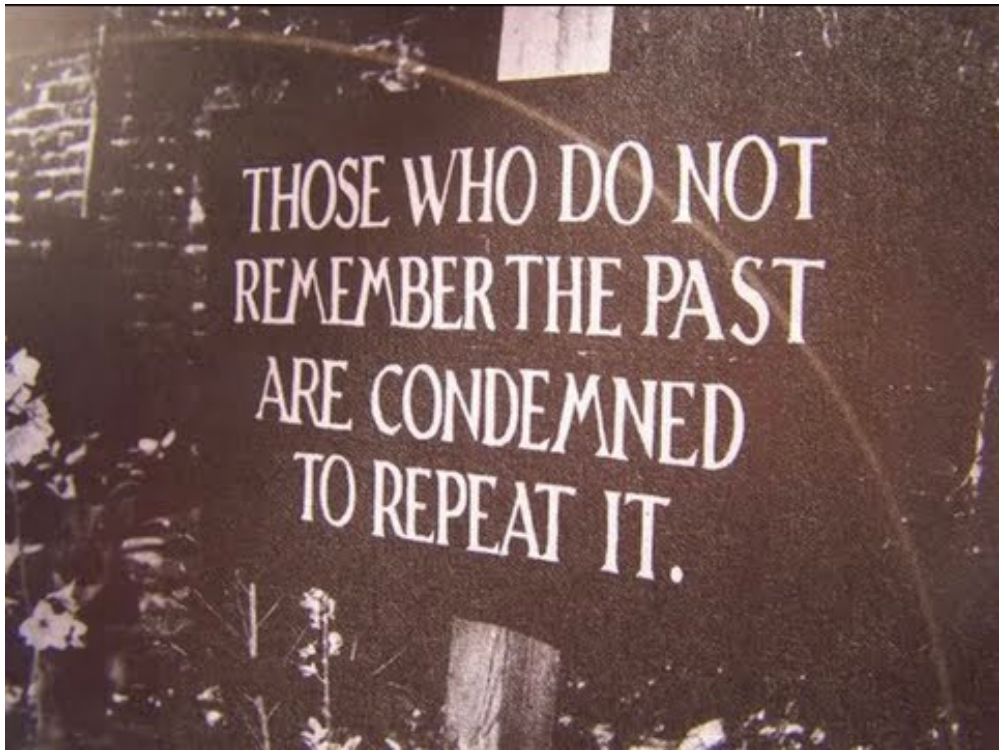
Algorithm 2: `FindSolution`(j)

```
1 if j == 0 then
2 |   return ∅
3 else if w_j + M[p[j]] > M[j − 1] then
4 |   return {j} ⋃ FindSolution(p[j])
5 else
6 |   return FindSolution(j − 1)
```

- 时间复杂度：$O(n)$
  - 函数*FindSolution*()的调用次数不超过n

动态规划

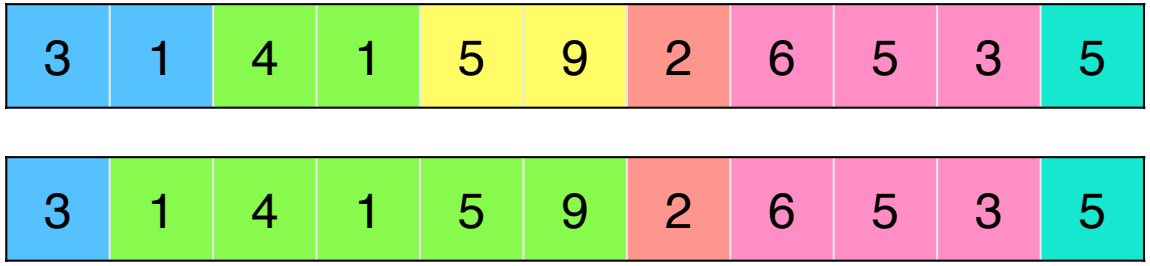
- 斐波那契数列
- 二项式系数
- 加权任务调度
- 文本分割
- 最长递增子序列
- 编辑距离
- 背包问题
- 最优二叉搜索树
- 树的最大独立集
- 动态规划的局限性



那些不能铭记过去的人注定要重蹈覆辙

文本分割

- 给定一串字符，能否将其分割成若干单词？
- 辅助函数$is_word(i, j)$
  - 给定的字符串存储在数组$s[1..n]$中
  - 如果$s[i..j]$是一个单词，返回$True$，否则返回$False$
- 问题：31415926535能否分割成一个质数单词序列？



使用多元选择构造递归

- 给定字符串$s[1..n]$，令解是O_n（若干子串），其值是$Splittable(n)$
- 如果$Splittable(n) = True$，那么下面情况之一必然成立
  - 情况1：$is\_word(1,n) = True$
  - 情况2：$is_word(2,n) = True$并且$Splittable(1) = True$
  - …
  - 情况n：$is_word(n,n) = True$并且$Splittable(n-1) = True$
- $Splittable(0) = True$：待分割的字符串为空，将其分割成0个单词
- 递归式

$$Splittable(n) = \begin{cases} True & \text{if } n = 0 \\ \vee_{j=1}^{n} \left(is\_word(j,n) \wedge Splittable(j-1)\right) & \text{otherwise} \end{cases}$$

自底向上的动态规划

$$Splittable(n) = \begin{cases} True & \text{if } n = 0 \\ \vee_{j=1}^{n} (is\_word(j,n) \wedge Splittable(j-1)) & \text{otherwise} \end{cases}$$

```
s = ' 3141592653'
n = len(s) - 1
M = [False] * (n + 1)

def splittable(n):
    M[0] = True
    for i in range(1, n + 1): #compute M[i]
        for j in range(1, i + 1):
            if is_word(s[j:i+1]) and M[j-1]:
                M[i] = True
                break
    return M[n]
```

自底向上的动态规划

	0	1	2	3	4	5	6	7	8	9	10
M	T	T	T	F	T	T	T	T	F	F	T
		3	31	314	3141	31415	314159	3141592	31415926	314159265	3141592653
				14	141	1415		141592	1415926	14159265	141592653
				4	41	415		41592	415926	4159265	41592653
						15		1592	15926	159265	1592653
						5		592	5926	59265	592653
								92	926	9265	92653
								2	26	265	2653
									6	65	653
										5	

- 时间复杂度：*is_word()*函数的调用次数
  - $O(n^2)$

```
def splittable(n):
    M[0] = True
    for i in range(1, n + 1): #compute M[i]
        for j in range(1, i + 1):
            if is_word(s[j:i+1]) and M[j-1]:
                M[i] = True
                break
    return M[n]
```

3141592653 5

构造最优解

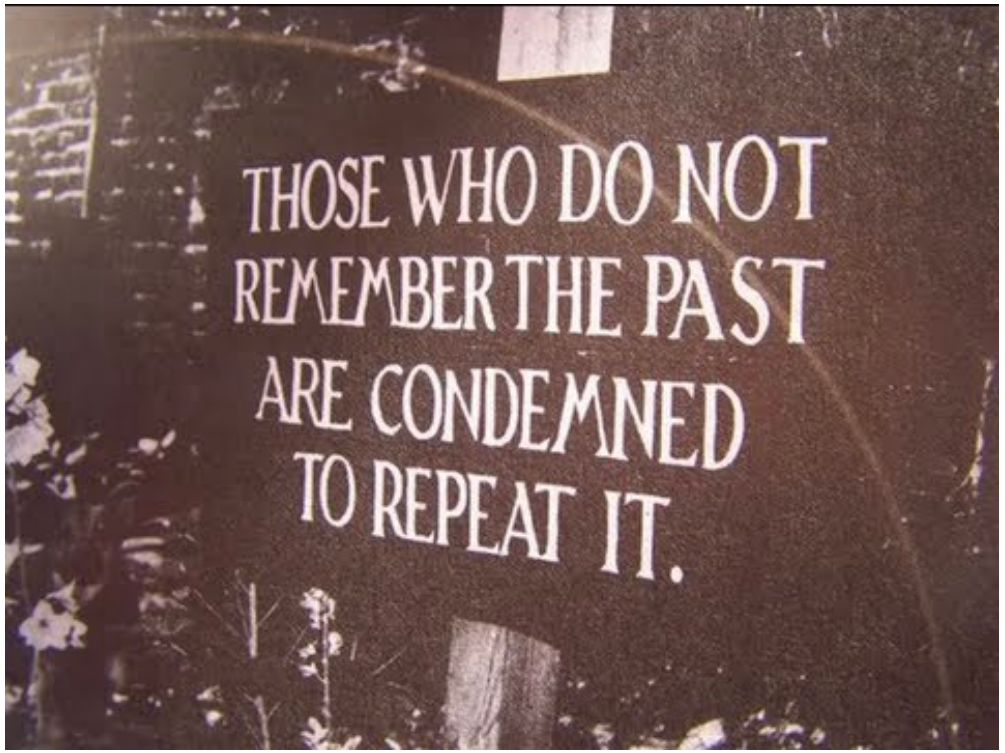
M	0	1	2	3	4	5	6	7	8	9	10
	T	T	T	F	T	T	T	T	F	F	T
		3	31	314	3141	31415	314159	3141592	31415926	314159265	3141592653
				14	14	1415		141592	1415926	14159265	141592653
				4	41	415		41592	415926	4159265	41592653
						15		1592	15926	159265	1592653
						5		592	5926	59265	592653
								92	926	9265	92653
								2	26	265	2653
									6	65	653
										5	

- 时间复杂度：$O(n)$

```
def find_solution(j):
    if M[j]:
        for i in range(j-1, -1, -1):
            if M[i] and is_word(s[i+1:j+1]):
                return find_solution(i) + [s[i+1:j+1]]
    return []
```

动态规划

- 斐波那契数列
- 二项式系数
- 加权任务调度
- 文本分割
- 最长递增子序列
- 编辑距离
- 背包问题
- 最优二叉搜索树
- 树的最大独立集
- 动态规划的局限性



那些不能铭记过去的人注定要重蹈覆辙

最长递增子序列

- 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：剩余元素的顺序并不改变）
  - backtracking的子序列包括：bring, tracking, 空串， backtracking
- 子串：如果一个子序列的所有元素在原始串中是连续的，那么它就是一个子串，比如tracking
- 给定一个整数序列，找出其中最长的递增子序列

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

使用二元选择构造递归关系

- 给定整数序列$s[1..n]$，令最长递增子序列是O_n，其长度是$OPT(n)$
- 对于O_n，下面两种情况之一必然成立
  - O_n不包含整数$s[n]$，此时，$OPT(n) = OPT(n-1)$
  - O_n包含整数$s[n]$，此时，$OPT(n)$与$OPT(i)(1 \le i \le n)$的关系是
    - 没有明显成立的关系
    - 更糟的是，仅仅根据$OPT(i)$无法确定O_n能否包含整数$s[n]$！
- 因此，仅用一个变量来刻画子问题无法构造递归关系
- 在回溯法中，第i步决定是否将$s[i]$放入子序列
  - 能否将$s[i]$放入子序列取决于最后一个放入子序列的元素

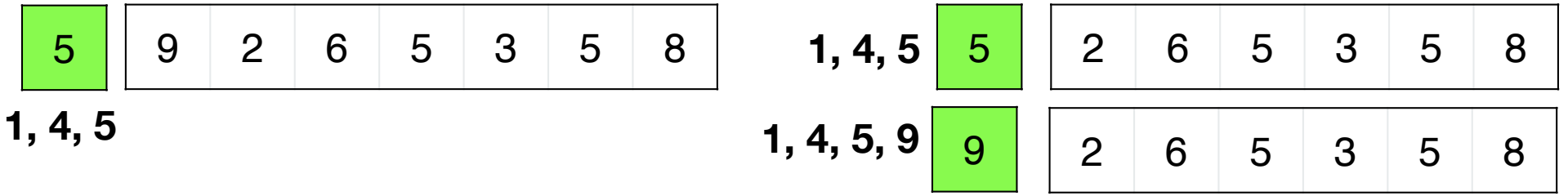
刻画子问题的另一个变量

使用二元选择构造递归关系

- 给定整数序列$s[1..n]$和整数$i < j$，令$OPT(i,j)$是$s[j..n]$中满足如下条件的最长递增子序列的长度：该子序列中的每个元素都大于$s[i]$
- 令$s[0] = -\infty$，原问题即为求解$OPT(0,1)$
- 如果$j > n$，不存在满足条件的子序列，所以$OPT(i,j) = 0$
- 否则，基于$s[j]$与$s[i]$的关系来决定最优子序列是否包含$s[j]$
  - 如果$s[i] \geq s[j]$，$s[j]$不满足加入子序列的条件，所以$OPT(i,j) = OPT(i,j+1)$

使用二元选择构造递归关系

- 令$OPT(i,j)$是$s[j..n]$中满足如下条件的最长递增子序列的长度：该子序列中的每个元素都大于$s[i]$
- 令$s[0] = -\infty$，原问题即为求解$OPT(0,1)$
- 如果$j > n$，不存在满足条件的子序列，所以$OPT(i,j) = 0$
- 否则，基于$s[j]$与$s[i]$的关系来决定最优子序列是否包含$s[j]$
  - 如果$s[i] < s[j]$，那么$OPT(i,j) = \max \begin{cases} OPT(i,j+1) \\ 1 + OPT(j,j+1) \end{cases}$
    - 最优子序列不包含$s[j]$：$OPT(i,j) = OPT(i,j+1)$
    - 最优子序列包含$s[j]$：$OPT(i,j) = 1 + OPT(j,j+1)$



使用二元选择构造递归关系

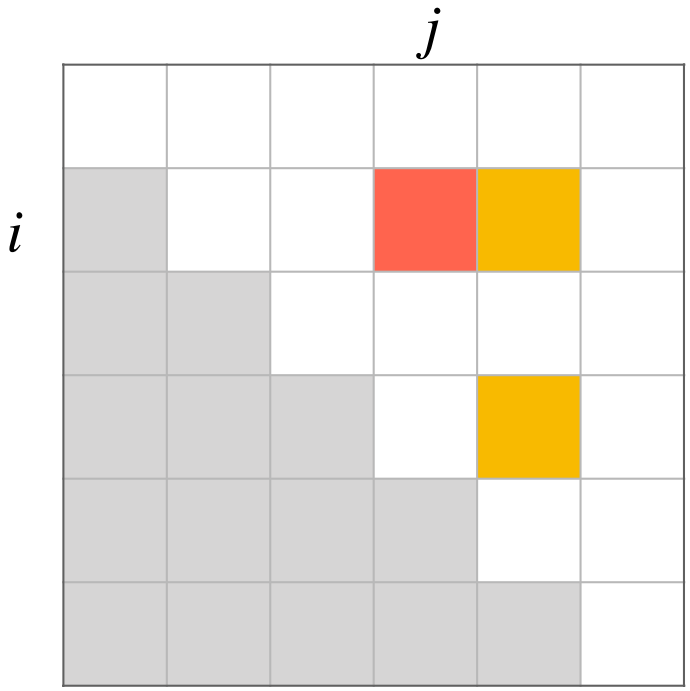
- 令$OPT(i,j)$是$s[j..n]$中满足如下条件的最长递增子序列的长度：该子序列中的每个元素都大于$s[i]$
- 令$s[0] = -\infty$，原问题即为求解$OPT(0,1)$
- 递归关系

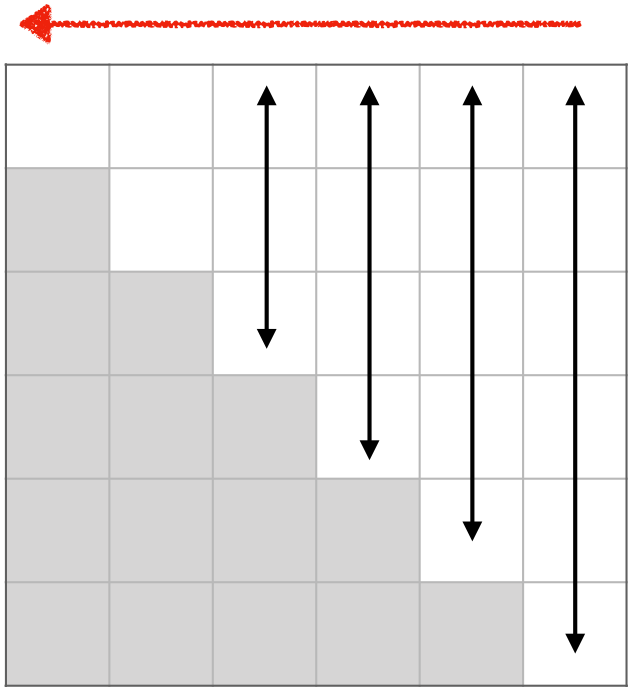
$$OPT(i,j) = \begin{cases} 0 & \text{if } j > n \\ OPT(i,j+1) & \text{if } s[i] \geq s[j] \\ \max \begin{cases} OPT(i,j+1) \\ 1 + OPT(j,j+1) \end{cases} & \text{otherwise} \end{cases}$$

自底向上的动态规划

- 递归关系

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





自底向上的动态规划

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

$n = 4$

$-\infty$	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

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

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

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

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

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

自底向上的动态规划

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

```python
M = [[None] * (n + 2) for _ in range(n + 2)]
def opt():
    for i in range(n+2):
        M[i][n+1] = 0
    for j in range(n, 0, -1):
        for i in range(j+1):
            if s[i] < s[j]:
                M[i][j] = max(M[i][j+1], 1 + M[j][j+1])
            else:
                M[i][j] = M[i][j+1]
    return M[0][1]
```

时间复杂度： $O(n^2)$ **空间复杂度：** $O(n^2) \longrightarrow O(n)$

构造最优解

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

$n = 4$

```
def find_solution(i, j):
    if j > n:
        return []
    if s[i] < s[j]:
        if M[i][j] == 1 + M[j][j+1]: #take s[j]
            return [s[j]] + find_solution(j, j+1)
        else: #skip s[j]
            return find_solution(i, j+1)
    else: #skip s[j]
        return find_solution(i, j+1)
```

时间复杂度：$O(n)$

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

$$OPT(0,1) = \max \begin{cases} OPT(0,2) \\ 1 + OPT(1,2) \end{cases} \quad take\ s[1]$$

$$OPT(1,2) = \max \begin{cases} OPT(1,3) \\ 1 + OPT(2,3) \end{cases} \quad skip\ s[2]$$

$$OPT(1,3) = \max \begin{cases} OPT(1,4) \\ 1 + OPT(3,4) \end{cases} \quad take\ s[3]$$

$$OPT(3,4) = \max \begin{cases} OPT(3,5) \\ 1 + OPT(4,5) \end{cases} \quad take\ s[4]$$

使用多元选择构造递归关系

- 给定整数序列$s[1..n]$，令$OPT(i)$是$s[i..n]$中满足如下条件的最长递增子序列的长度：子序列必须以$s[i]$开始
- 令$s[0] = -\infty$，原问题即为求解$OPT(0) - 1$
- 在O_i中，$s[i]$的下一个元素必然是下列情况中的一种
  - $\{s[j] \mid j > i \text{ and } s[j] > s[i]\}$
- 递归关系

$$OPT(i) = 1 + \max\{OPT(j) \mid j > i \text{ and } s[j] > s[i]\}$$

其中，$\max \varnothing = 0 \Rightarrow OPT(n) = 1$

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

自底向上的动态规划

- 递归关系

$$OPT(i) = 1 + \max\{OPT(j) \mid j > i \text{ and } s[j] > s[i]\}$$

其中，$\max \varnothing = 0 \Rightarrow OPT(n) = 1$

```
M = [1] * (n + 1)
def opt_a():
    for i in range(n-1, -1, -1):
        for j in range(i+1, n+1):
            if s[i] < s[j]:
                M[i] = max(M[i], 1 + M[j])
    return M[0] - 1
```

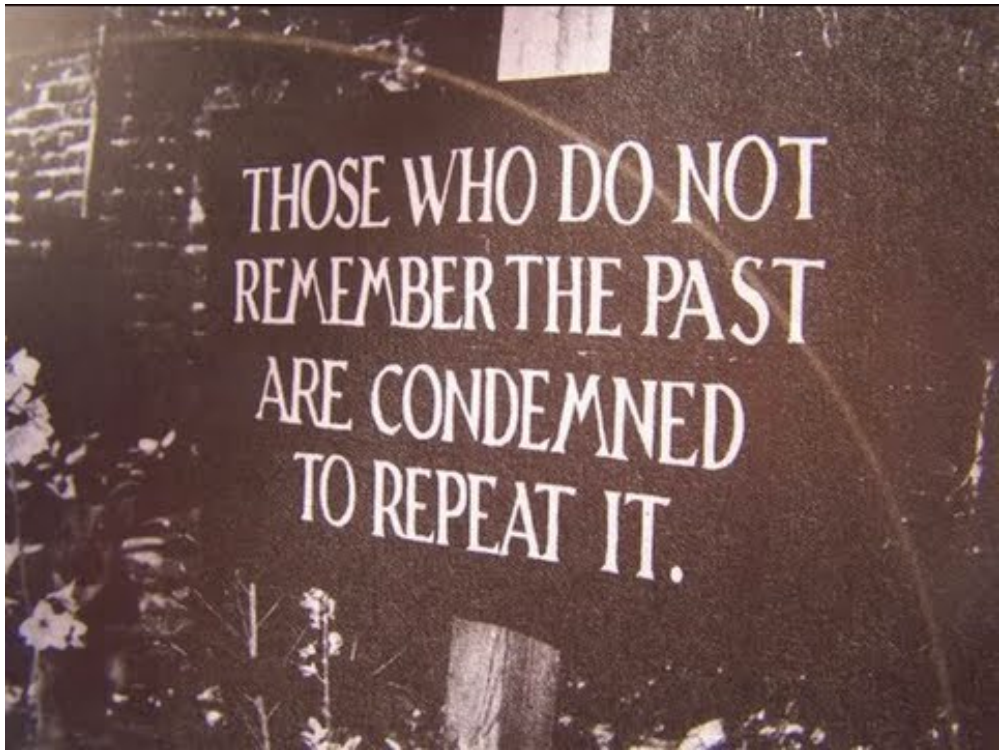
时间复杂度： $O(n^2)$

空间复杂度： $O(n)$

s	$-\infty$	3	1	4	1	5	9	2	6	5	3	5	8
M	6	5	5	4	5	3	1	4	2	2	3	2	1

动态规划

- 斐波那契数列
- 二项式系数
- 加权任务调度
- 文本分割
- 最长递增子序列
- 编辑距离
- 背包问题
- 最优二叉搜索树
- 树的最大独立集
- 动态规划的局限性



那些不能铭记过去的人注定要重蹈覆辙

编辑距离

- 两个字符串之间的编辑距离是将一个字符串转换为另一个字符串所需的最少的字母插入、删除和替换次数
  - FOOD $\overset{sub}{\rightarrow}$ MOOD $\overset{sub}{\rightarrow}$ MOND $\overset{ins}{\rightarrow}$ MONED $\overset{sub}{\rightarrow}$ MONEY
- 使用序列比对来对编辑过程进行可视化
  - 第一行的空格表示插入
  - 第二行的空格表示删除
  - 具有不同字符的列表示替换
  - 编辑距离 = 具有不同字符的列数
- 给定字符串A和B，计算它们的编辑距离

F	O	O		D
M	O	N	E	Y

构造递归关系

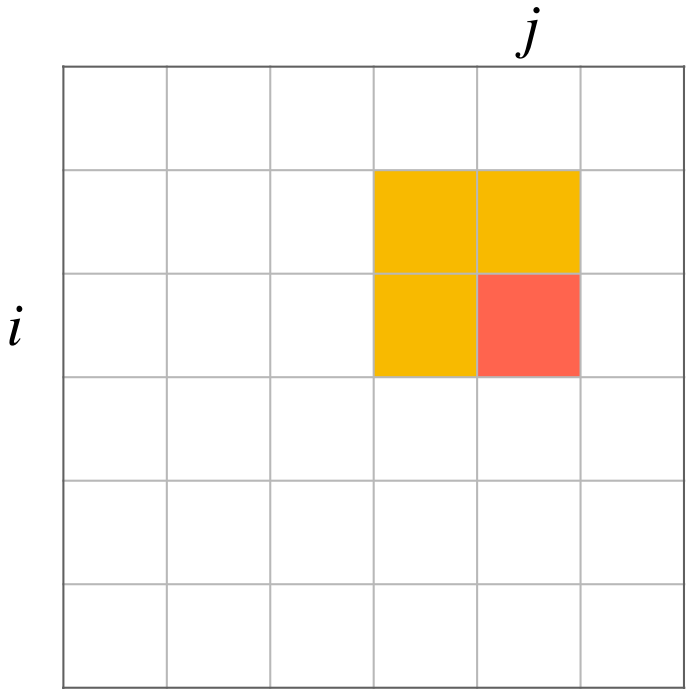
F	O	O		
M	O	N	E	

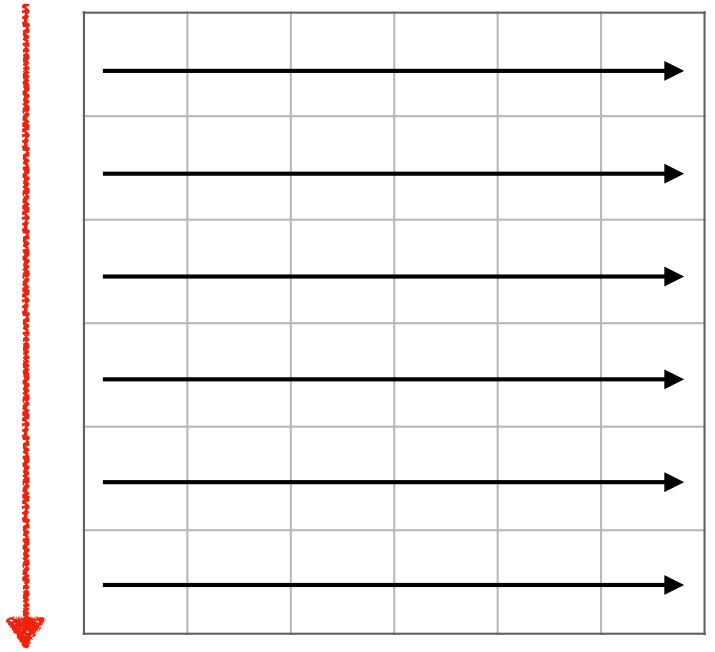
- 最优子结构性质
  - 删除最后一列，剩余列必然代表前缀的编辑距离
- 给定字符串$A[1..m]$和$B[1..n]$，令$OPT(i,j)$是前缀$A[1..i]$和$B[1..j]$的编辑距离
  - 原问题即为求解$OPT(m,n)$
- 当$i>0,\ j>0$时，$A[1..i]$和$B[1..j]$比对的最后一列必然是下列三种情况之一
  - 第一行最后一列为空格：$OPT(i,j)=OPT(i,j-1)+1$
  - 第二行最后一列为空格：$OPT(i,j)=OPT(i-1,j)+1$
  - 两行最后一列都不为空格：$OPT(i,j)=OPT(i-1,j-1)+[A[i]\neq B[j]]$
- 当$i=0$或者$j=0$时
  - $OPT(0,j)=j$：空串经过j次插入得到长度为j的字符串
  - $OPT(i,0)=i$：空串经过i次插入得到长度为i的字符串

自底向上的动态规划

- 递归关系

$$OPT(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} OPT(i,j-1)+1 \\ OPT(i-1,j)+1 \\ OPT(i-1,j-1)+[A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$





自底向上的动态规划

- 递归关系

$$OPT(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} OPT(i, j-1) + 1 \\ OPT(i-1, j) + 1 \\ OPT(i-1, j-1) + [A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$

```
def opt(): #M[0..m][0..n]
    for j in range(n + 1): #i=0
        M[0][j] = j
    for i in range(m + 1): #j=0
        M[i][0] = i
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            insert = M[i][j-1] + 1
            delete = M[i-1][j] + 1
            sub = M[i-1][j-1] + (1 if A[i] != B[j] else 0)
            M[i][j] = min(insert, delete, sub)
    return M[m][n]
```

自底向上的动态规划

		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	0	1	2	3	4	5	6	7	8
L	2	1	0	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	4	5	6
R	4	3	2	2	2	2	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	3	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	4	5	6
I	9	8	7	7	7	7	6	5	5	6
C	10	9	8	8	8	8	7	6	6	6

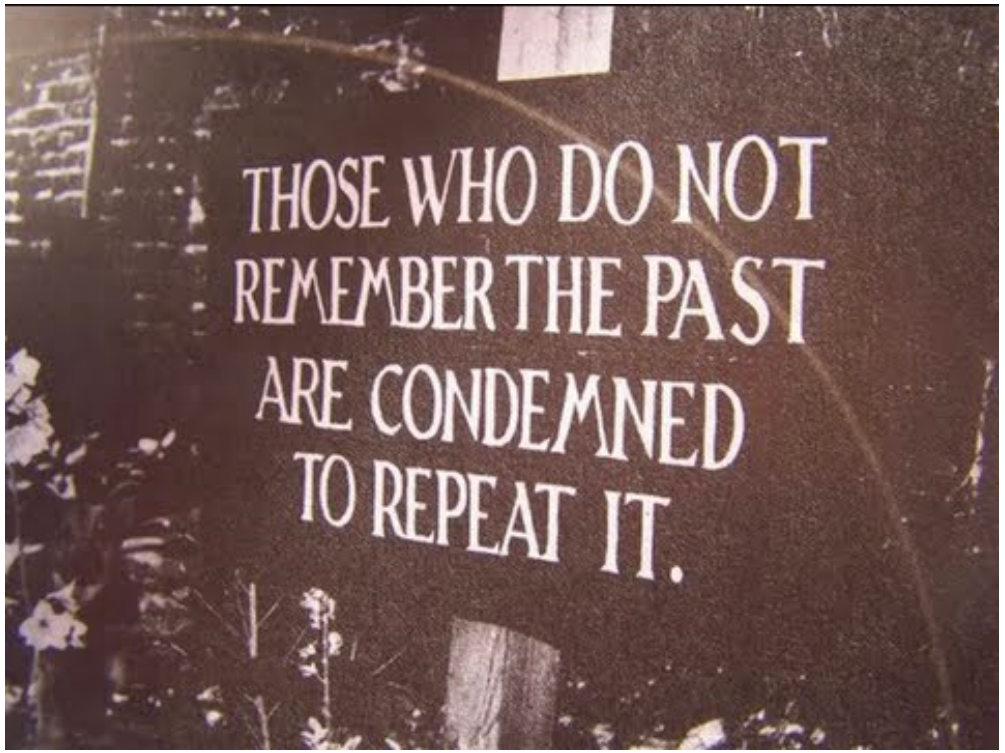
A	L	T	R	U	I	S	T	I	C
A	L	G	O	R	I		T	H	M

A	L		T	R	U	I	S	T	I	C
A	L	G	O	R		I		T	H	M

A	L	T		R	U	I	S	T	I	C
A	L	G	O	R		I		T	H	M

动态规划

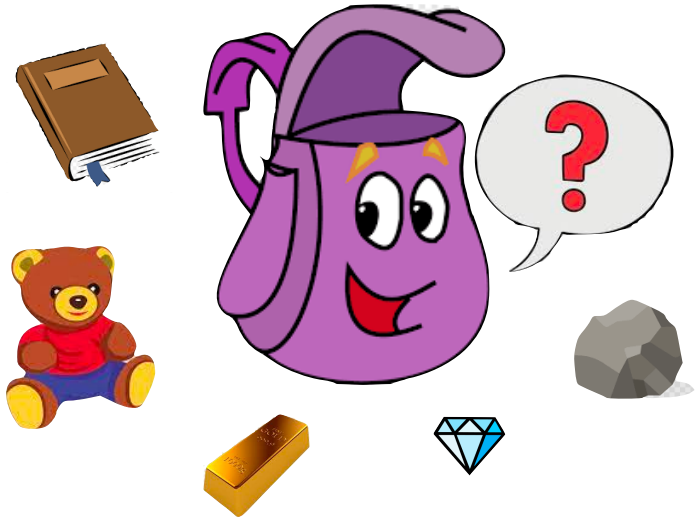
- 斐波那契数列
- 二项式系数
- 加权任务调度
- 文本分割
- 最长递增子序列
- 编辑距离
- 背包问题
- 最优二叉搜索树
- 树的最大独立集
- 动态规划的局限性



那些不能铭记过去的人注定要重蹈覆辙

背包问题

- 给定一组物品和一个背包，每种物品都有自己的重量和价值，如何在不超过背包最大承重的情况下选择物品放入背包使得总价值最高
  - n种物品，物品i的价值$v_i > 0$，重量$w_i > 0$，v_i和w_i都是整数
  - 背包的最大承重W，W也是整数
- 物品{1,5}的重量是15，价值是19
- 物品{1,2,3}的重量是9，价值是16
- 物品{1,3,4}的重量是14，价值是20



物品	重量	价值	单位价值
1	1	2	2
2	3	6	2
3	5	8	1.6
4	8	10	1.25
5	14	17	1.21

背包最大承重$W = 15$

构造递归关系

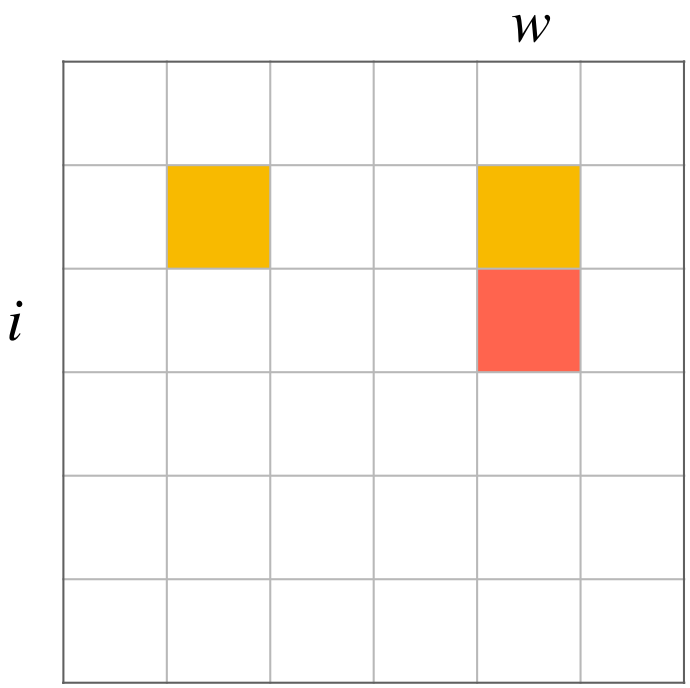
- 一个物品是否可以放入背包取决于背包的剩余容量
  - 子问题不仅与剩余物品有关，也与剩余容量有关
- 令$OPT(i, w)$表示当剩余物品为$\{1,2,\cdots, i\}$，背包剩余容量为w时的背包问题的最优解的值，即背包中物品的总价值
  - 如果$w_i > w$，物品i无法放入背包
    - $OPT(i, w) = OPT(i-1, w)$
  - 否则，可以放也可以不放
    - 不放：剩余物品为$\{1,2,\cdots, i-1\}$，背包剩余容量为w
    - 放：剩余物品为$\{1,2,\cdots, i-1\}$，背包剩余容量为$w - w_i$
    - $OPT(i, w) = \max\{OPT(i-1, w),\ v_i + OPT(i-1, w-w_i)\}$
  - 当$i = 0$时，$OPT(i, w) = 0$

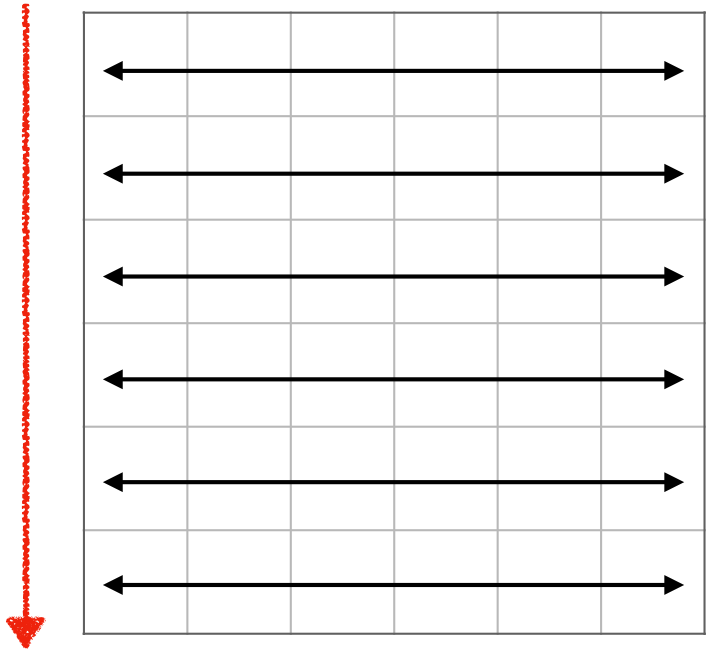
自底向上的动态规划

- 递归关系

$$OPT(i,w)=\begin{cases}0 & \text{if } i=0\\ OPT(i-1,w) & \text{if } w_i>w\\ \max\begin{cases}OPT(i-1,w)\\ v_i+OPT(i-1,w-w_i)\end{cases} & \text{otherwise}\end{cases}$$

- 原问题即为求解$OPT(n,W)$





自底向上的动态规划

- 递归关系

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max \begin{cases} OPT(i-1,w) \\ v_i + OPT(i-1,w-w_i) \end{cases} & \text{otherwise} \end{cases}$$

- 原问题即为求解$OPT(n,W)$

```
def opt():
    for j in range(W+1):
        M[0][j] = 0
    for i in range(1, n+1):
        for j in range(W+1):
            if w[i] > j:
                M[i][j] = M[i-1][j]
            else:
                M[i][j] = max(M[i-1][j], v[i] + M[i-1][j-w[i]])
    return M[n][W]
```

自底向上的动态规划

- 递归关系

$$OPT(i,w)=\begin{cases}0 & \text{if } i=0\\ OPT(i-1,w) & \text{if } w_i>w\\ \max\begin{cases}OPT(i-1,w)\\ v_i+OPT(i-1,w-w_i)\end{cases} & \text{otherwise}\end{cases}$$

物品	重量	价值
1	1	2
2	3	6
3	5	8
4	8	10
5	14	17

最优解是$\{1,3,4\}$，其值是20

如果$M[i][j]>M[i-1][j]$，选取物品i

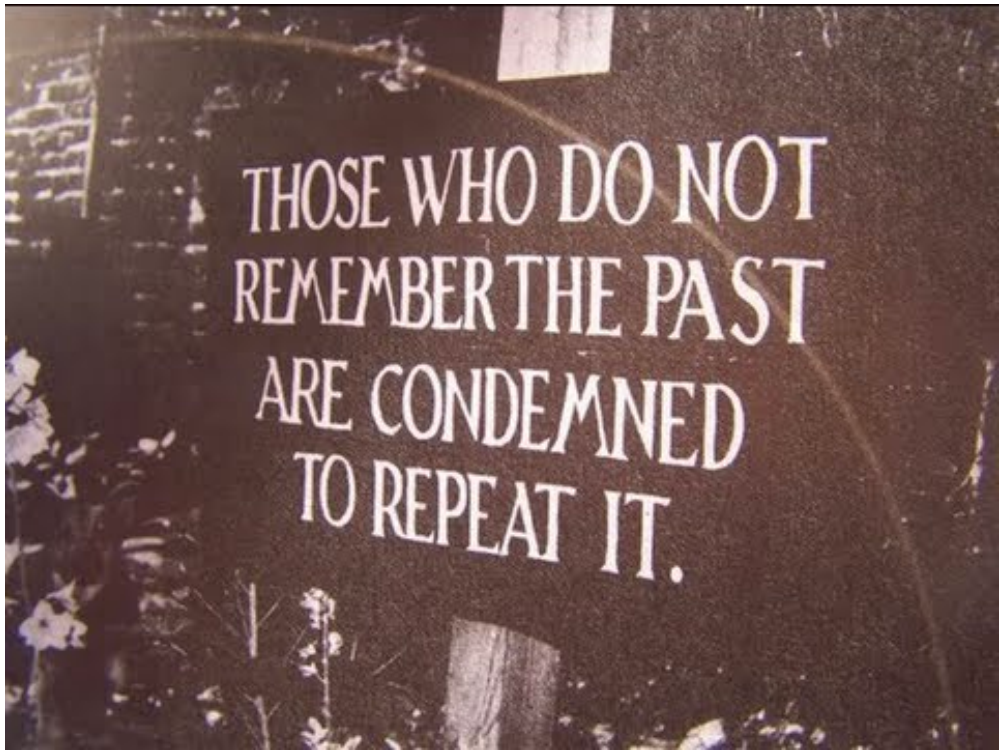
i \ W	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
{ }	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
{ 1, 2 }	0	2	2	6	8	8	8	8	8	8	8	8	8	8	8	8
{ 1, 2, 3 }	0	2	2	6	8	8	10	10	14	16	16	16	16	16	16	16
{ 1, 2, 3, 4 }	0	2	2	6	8	8	10	10	14	16	16	16	18	18	20	20
{ 1, 2, 3, 4, 5 }	0	2	2	6	8	8	10	10	14	16	16	16	18	18	20	20

动态规划算法的复杂度

- 时间复杂度：$O(nW)$
- 空间复杂度：$O(nW)$
- 回溯求解背包问题的时间复杂度
  - $O(2^n)$
- 针对背包问题，回溯和动态规划哪个更好？
  - 取决于W和2^n的值
  - 当$W \gg 2^n$时，动态规划算法更慢！
    - 动态规划考虑了大量回溯没有考虑的子问题
- $O(nW)$不是多项式时间
  - 伪多项式时间（pseudo-polynomial time）

动态规划

- 斐波那契数列
- 二项式系数
- 加权任务调度
- 文本分割
- 最长递增子序列
- 编辑距离
- 背包问题
- 最优二叉搜索树
- 树的最大独立集
- 动态规划的局限性



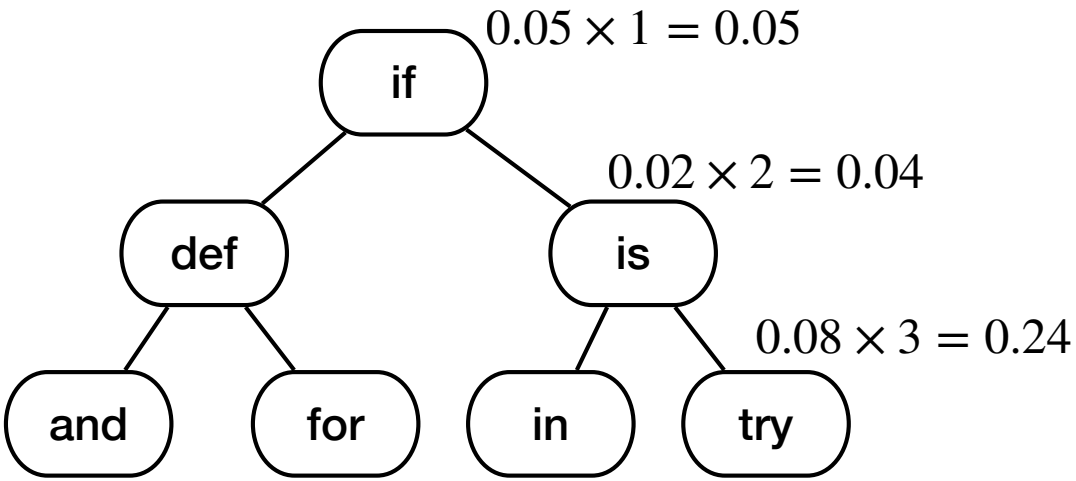
那些不能铭记过去的人注定要重蹈覆辙

最优二叉搜索树

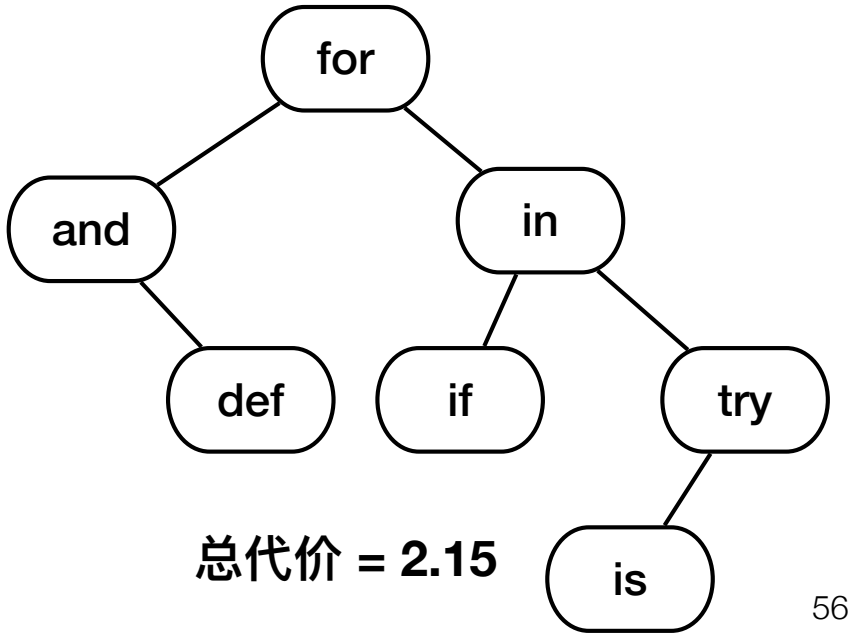
- 给定n个节点$v_1, \cdots, v_n$，它们具有不同的搜索概率：$f[1..n]$

and	def	for	if	in	is	try
0.22	0.18	0.20	0.05	0.25	0.02	0.08

- 节点v_i的搜索代价$c(v_i) = d(v_i) + 1$
- 二叉搜索树T的总搜索代价$Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] \cdot c(v_i)$



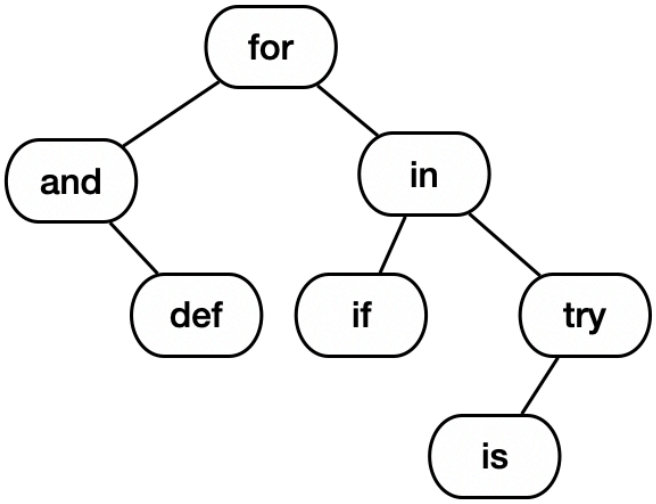
总代价 = 2.7

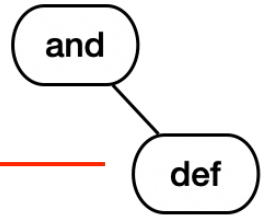


总代价 = 2.15

搜索代价的递归定义

- 二叉搜索树T的总搜索代价$Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] \cdot c(v_i)$



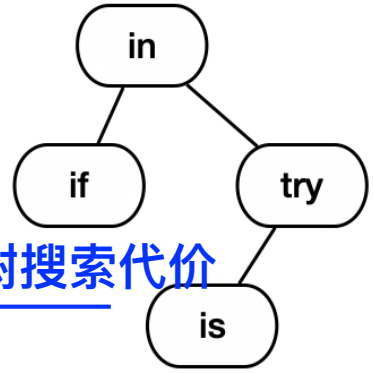


左子树搜索代价

$f[1] \cdot 1 + f[2] \cdot 2$

总搜索代价

$f[1] \cdot 2 + f[2] \cdot 3 +$
$f[3] \cdot 1 +$
$f[4] \cdot 3 + f[5] \cdot 2 + f[6] \cdot 4 + f[7] \cdot 3$



右子树搜索代价

$f[4] \cdot 2 + f[5] \cdot 1 + f[6] \cdot 3 + f[7] \cdot 2$

$$Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] + Cost(left(T), f[1..r-1]) + Cost(right(T), f[r+1..n])$$

构造递归关系

$$Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] + Cost(left(T), f[1..r-1]) + Cost(right(T), f[r+1..n])$$

- 假设最优二叉搜索树T_{opt}的根节点是v_r，那么
  - $left(T_{opt})$必然是节点$v_1, \cdots, v_{r-1}$的最优二叉搜索树
  - $right(T_{opt})$必然是节点$v_{r+1}, \cdots, v_n$的最优二叉搜索树
- 令$OptCost(i,k)$是节点$v_i, \cdots, v_k$的最优二叉搜索树的搜索代价
  - 原始问题即为求解$OptCost(1,n)$

$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ \sum_{j=i}^{k} f[j] + \min_{i \le r \le k}\{OptCost(i, r-1) + OptCost(r+1, k)\} & \text{otherwise} \end{cases}$$

简化递归关系

$$OptCost(i,k)=\begin{cases}0 & \text{if } i>k\\ \sum_{j=i}^{k} f[j]+\min_{i\le r\le k}\{OptCost(i,r-1)+OptCost(r+1,k)\} & \text{otherwise}\end{cases}$$

- 令$F(i,k)=\sum_{j=i}^{k} f[j]$，其中$i\le k$，显然有

$$F(i,k)=\begin{cases}f[i] & \text{if } i=k\\ F(i,k-1)+f[k] & \text{otherwise}\end{cases}$$

- 事先计算所有的$F(i,k)$，时间复杂度$O(n^2)$
  - 动态规划

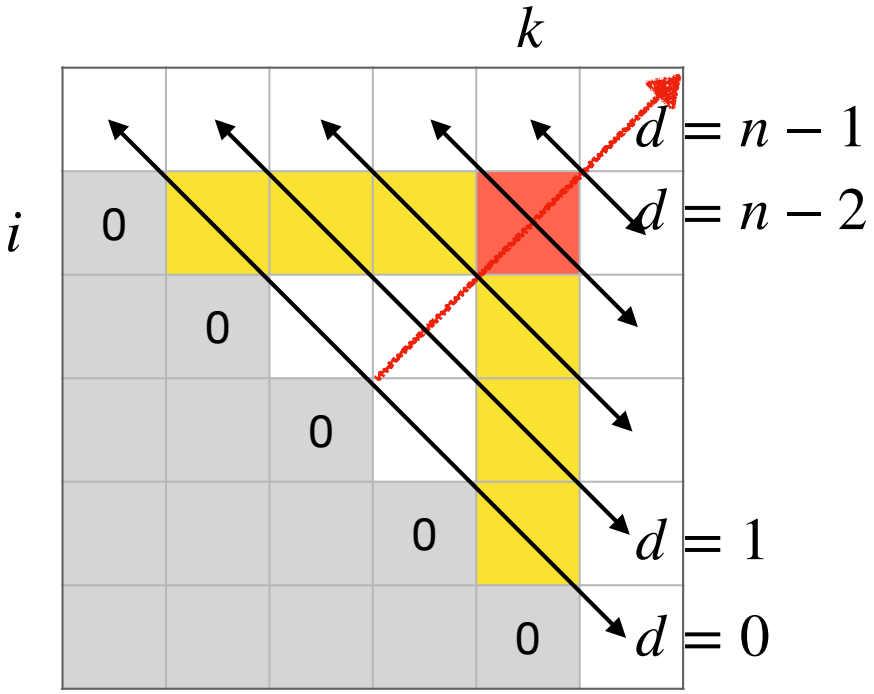
```
F = [[None] * (n + 1) for _ in range(n + 1)]


def compute_freq():
    for i in range(1, n + 1):
        F[i][i-1] = 0
        for k in range(i, n + 1):
            F[i][k] = F[i][k-1] + freq[k]
```

自底向上的动态规划 - 对角线优先

- 递归关系

$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ F(i,k) + \min_{i \le r \le k}\{OptCost(i,r-1) + OptCost(r+1,k)\} & \text{otherwise} \end{cases}$$

n条对角线，第d条上有$n-d$个元素




```
M = [[None] * (n + 1) for _ in range(n + 2)]
```

```
def opt():
    for i in range(1, n + 2): # why i = n+1?
        M[i][i-1] = 0
    for d in range(0, n):
        for i in range(1, n - d + 1):
            compute_opt_cost(i, i + d)
    return M[1][n]
```

```
def compute_opt_cost(i, k):
    M[i][k] = F[i][k] + min(M[i][r-1] + M[r+1][k] for r in range(i, k + 1))
```

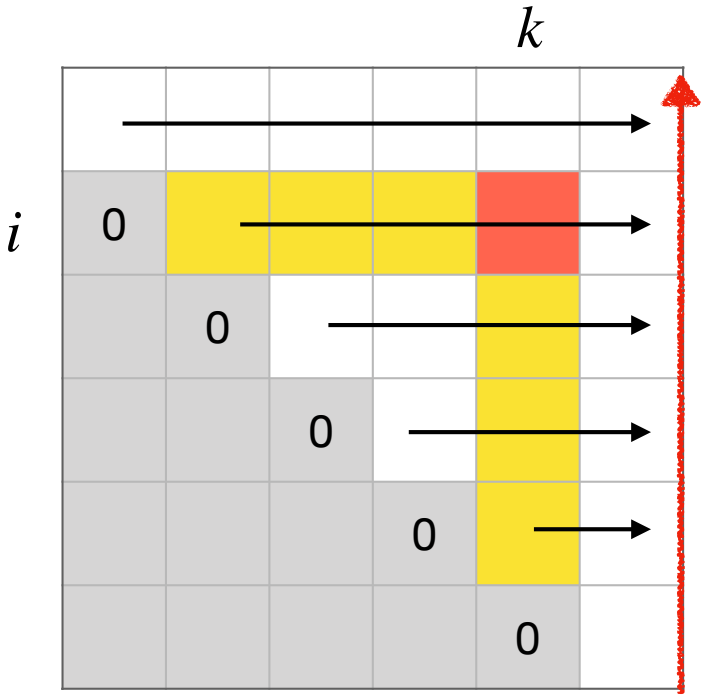
and	def	for	if	in	is	try
0.22	0.18	0.20	0.05	0.25	0.02	0.08

	0	1	2	3	4	5	6	7
0								
1	0	0.22	0.58	1.02	1.17	1.83	1.89	2.15
2		0	0.18	0.56	0.66	1.21	1.27	1.53
3			0	0.2	0.3	0.8	0.84	1.02
4				0	0.05	0.35	0.39	0.57
5					0	0.25	0.29	0.47
6						0	0.02	0.12
7							0	0.08
8								0

自底向上的动态规划 - 行优先

- 递归关系

$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ F(i,k) + \min_{i \le r \le k}\{OptCost(i,r-1) + OptCost(r+1,k)\} & \text{otherwise} \end{cases}$$

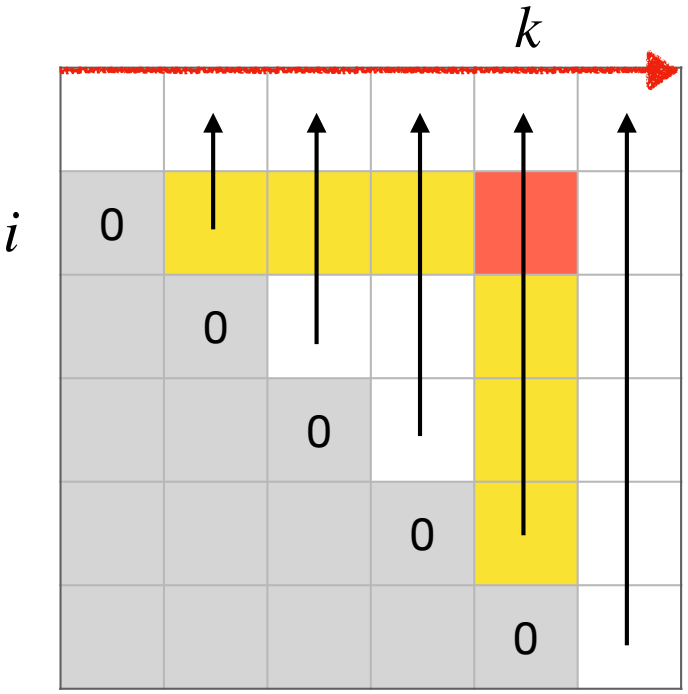



```
def opt_v1():
    for i in range(1, n + 2): # why i = n+1?
        M[i][i-1] = 0
    for i in range(n, 0, -1):
        for j in range(i, n + 1):
            compute_opt_cost(i, j)
    return M[1][n]
```

自底向上的动态规划 - 列优先

- 递归关系

$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ F(i,k) + \min_{i \le r \le k}\{OptCost(i,r-1) + OptCost(r+1,k)\} & \text{otherwise} \end{cases}$$




```
def opt_v2():
    for i in range(1, n + 2): # why i = n+1?
        M[i][i-1] = 0
    for j in range(1, n + 1):
        for i in range(j, 0, -1):
            compute_opt_cost(i, j)
    return M[1][n]
```

构造最优解

- 递归关系

$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ F(i,k) + \min_{i \le r \le k}\{OptCost(i,r-1) + OptCost(r+1,k)\} & \text{otherwise} \end{cases}$$

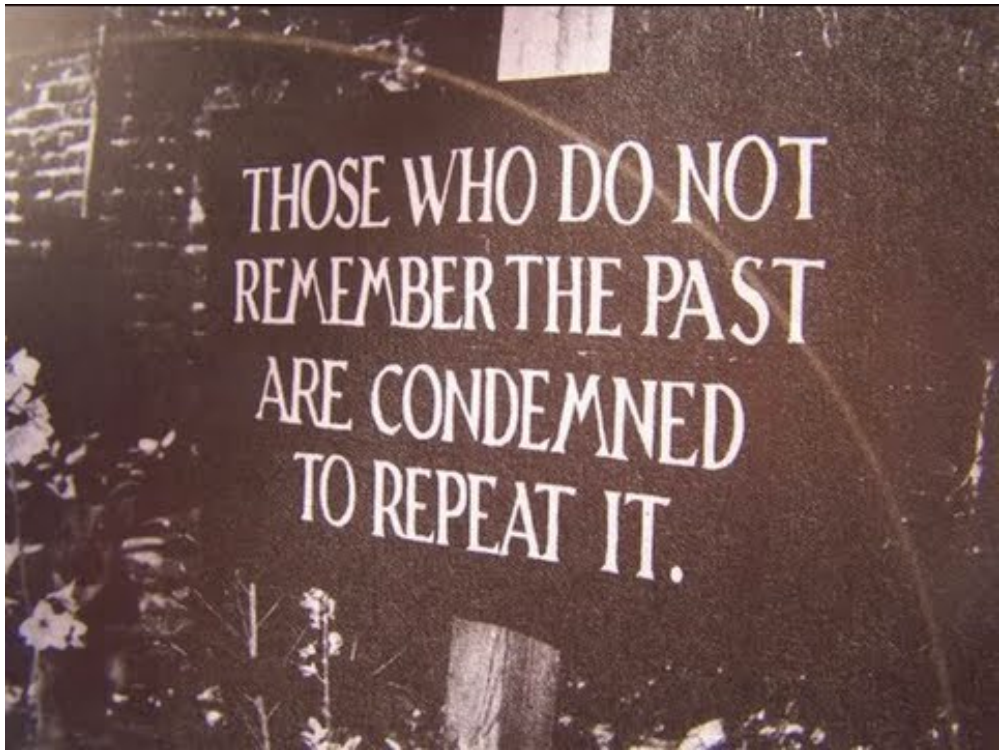
```
def find_solution(i, k):
    if i > k:
        return []
    elif i == k:
        return [i]
    else:
        b = i
        c = M[i][i-1] + M[i+1][k]
        for r in range(i+1, k+1):
            t = M[i][r-1] + M[r+1][k]
            if c > t:
                c = t
                b = r
        return [b, find_solution(i, b-1), find_solution(b+1, k)]
```

算法复杂度

- 时间复杂度：$O(n^3)$
  - 事先计算$F(i,k)$：$O(n^2)$
  - compute_opt_cost()：$O(n)$
  - 无论哪种计算顺序，compute_opt_cost()的调用次数都是$O(n^2)$
- 空间复杂度：$O(n^2)$
- 构造最优解的时间复杂度
  - $O(n^2)$

动态规划

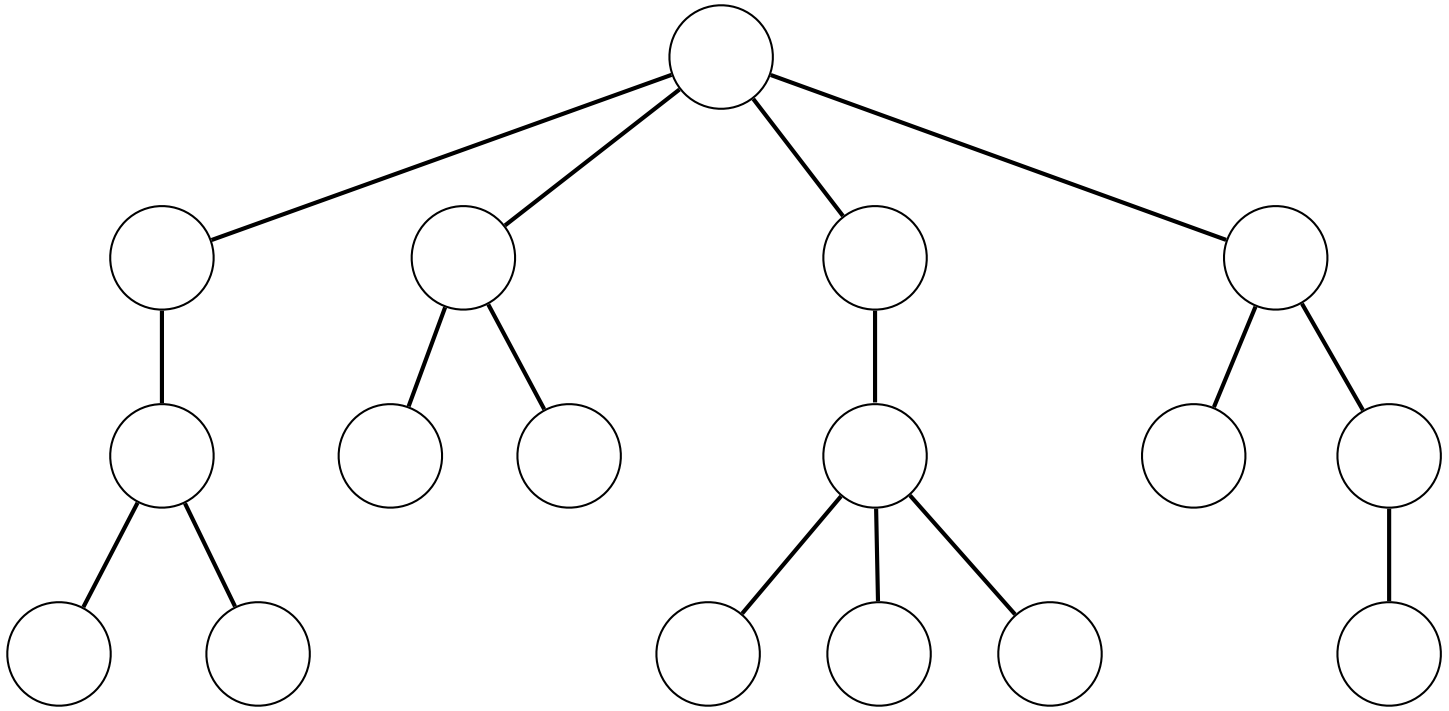
- 斐波那契数列
- 二项式系数
- 加权任务调度
- 文本分割
- 最长递增子序列
- 编辑距离
- 背包问题
- 最优二叉搜索树
- 树的最大独立集
- 动态规划的局限性



那些不能铭记过去的人注定要重蹈覆辙

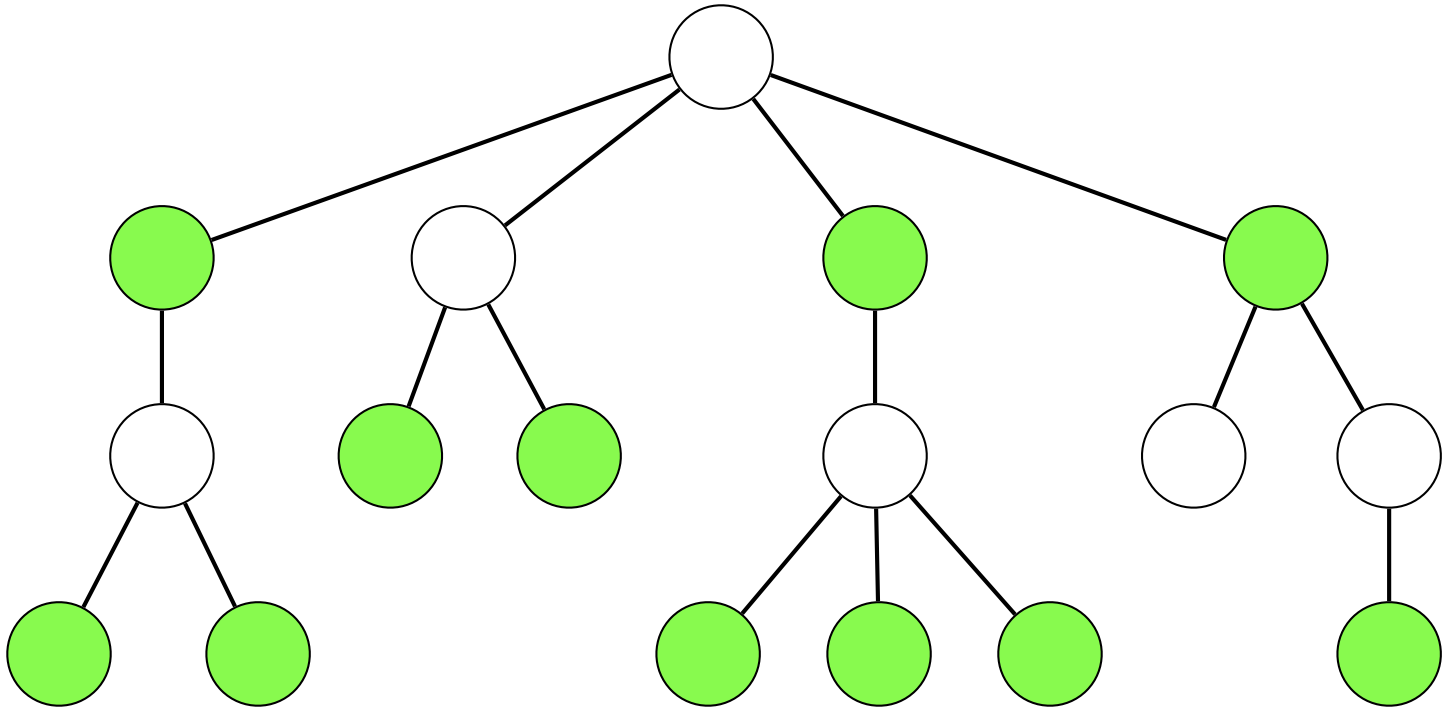
树的最大独立集

- 独立集：图中两两不相邻顶点构成的集合
- 最大独立集：不是任何其他独立集子集的独立集
- NP难问题，但对于某些特殊的图，存在多项式时间的算法
- 目标：计算树的最大独立集



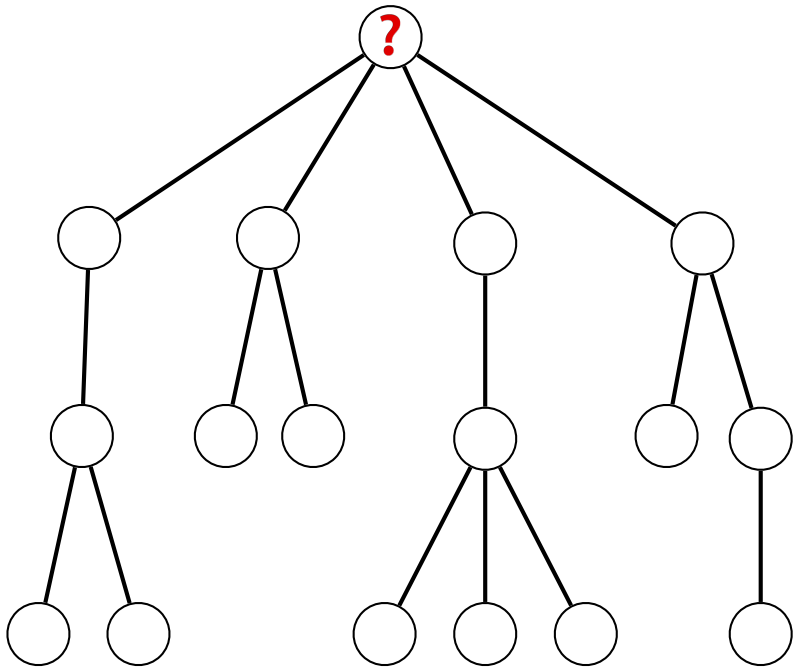
树的最大独立集

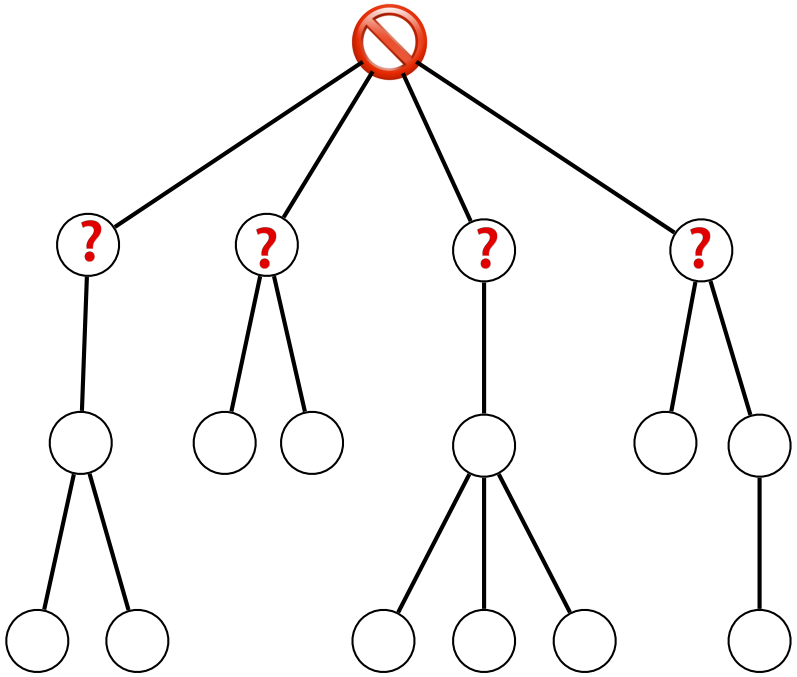
- 独立集：图中两两不相邻顶点构成的集合
- 最大独立集：不是任何其他独立集子集的独立集
- NP难问题，但对于某些特殊的图，存在多项式时间的算法
- 目标：计算树的最大独立集



构造递归关系

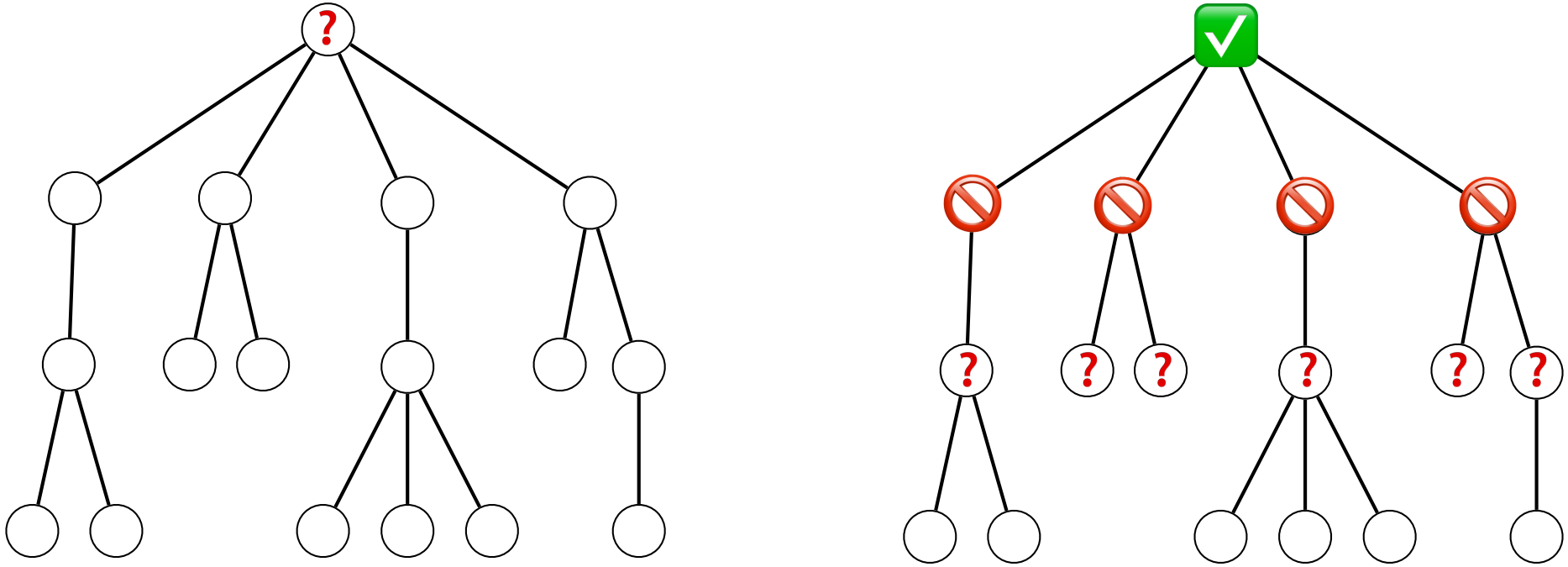
- $MIS(v)$：以v为根节点的树的最大独立集的大小
  - v不在该最大独立集中
    - 以v的孩子为根节点的最大独立集的并集





构造递归关系

- $MIS(v)$：以v为根节点的树的最大独立集的大小
  - v在该最大独立集中
    - v的孩子肯定不在该独立集中
    - 以v的孙子为根节点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的孩子和孙子节点
    - 树的先序遍历：先遍历树的所有子树，再访问树的根节点

算法实现

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

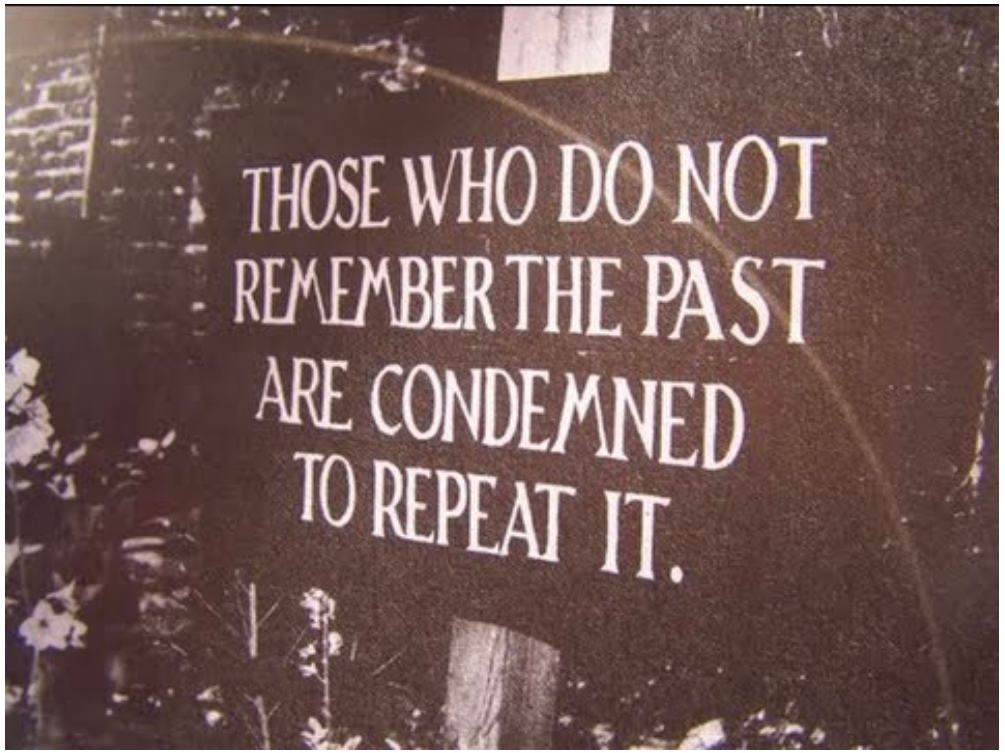
```
def tree_mis(v):
    skip_v = 0
    for child in v.children():
        skip_v += tree_mis(child)
    keep_v = 1
    for grandchild in v.grandchildren():
        keep_v += tree_mis(grandchild)
    v.mis = max(skip_v, keep_v)
    return v.mis
```

```
class Tree:
    def __init__(self, L): #initialize a new tree given a list of lists
        iterator = iter(L)
        self.__data = next(iterator) #L[0]
        self.__children = [Tree(c) for c in iterator] #L[i], if any
        self.__mis = None

    @property
    def data(self):
        return self.__data

    @property
    def mis(self):
        return self.__mis

    @mis.setter
    def mis(self, m):
        self.__mis = m
```

```
    def children(self):
        for child in self.__children:
            yield child

    def grandchildren(self):
        for child in self.__children:
            for grandchild in child.__children:
                yield grandchild
```

动态规划

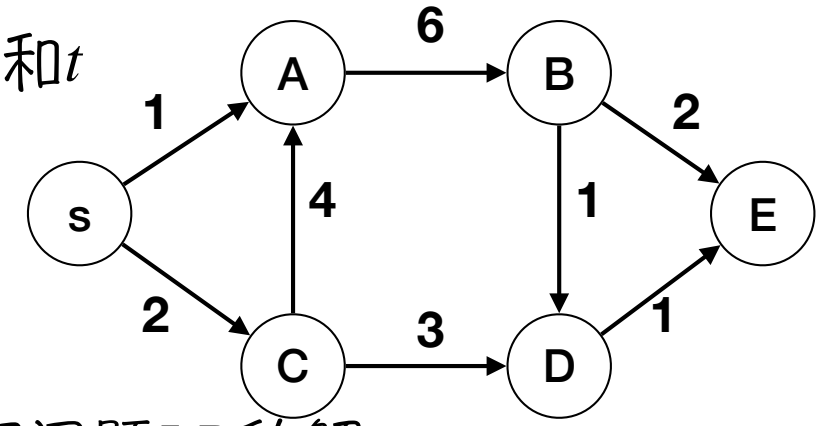
- 斐波那契数列
- 二项式系数
- 加权任务调度
- 文本分割
- 最长递增子序列
- 编辑距离
- 背包问题
- 最优二叉搜索树
- 树的最大独立集
- 动态规划的局限性



那些不能铭记过去的人注定要重蹈覆辙

有向无环图中的最长路径

- 给定有向无环图G及其中的两个顶点s和t
  - 求s到t的最长路径的长度
  - 该问题记为LP_t



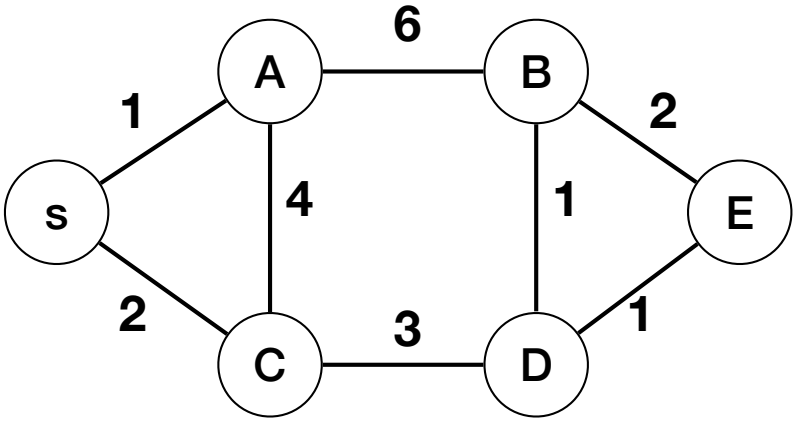
- 令$OPT(t)$是s到t的最长路径的长度，即问题LP_t的解
  - $OPT(D) = \max\{OPT(B) + 1,\ OPT(C) + 3\}$
- 递归关系

$$OPT(v) = \begin{cases} 0 & \text{if } v = s \\ \max_{u \to v}\{OPT(u) + w(u \to v)\} & \text{otherwise} \end{cases}$$

- 原问题即为求解$OPT(t)$
- 能否据此设计一个求解最长路径的动态规划算法🤯

无向图中的最长路径

- 给定无向图G及其中的两个顶点s和t
  - 求s到t的最长简单路径的长度
    - 路径中顶点不重复



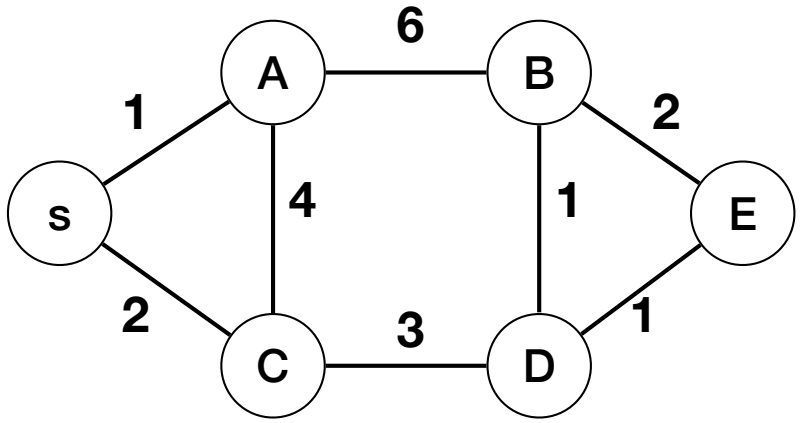
- 令$OPT(t)$是s到t的最长简单路径的长度
  - $OPT(D) = \max\{OPT(B) + 1,\ OPT(C) + 3,\ OPT(E) + 1\}$
- 递归关系

$$OPT(v) = \begin{cases} 0 & \text{if } v = s \\ \max_{u \to v}\{OPT(u) + w(u \to v)\} & \text{otherwise} \end{cases}$$

- 原问题即为求解$OPT(t)$
- 能否据此设计一个求解最长路径的动态规划算法🤯

无向图中的最长路径

- 给定无向图G及其中的两个顶点s和t
  - 求s到t的最长简单路径的长度
    - 路径中顶点不重复



- 令$OPT(t)$是s到t的最长简单路径的长度
  - $OPT(D) = \max\{OPT(B) + 1,\ OPT(C) + 3,\ OPT(E) + 1\}$
- 递归关系

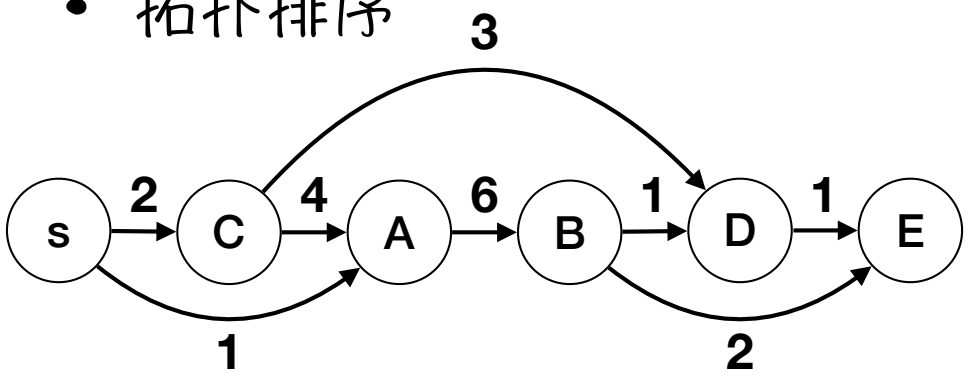
$$OPT(v) = \begin{cases} 0 & \text{if } v = s \\ \max_{u \to v}\{OPT(u) + w(u \to v)\} & \text{otherwise} \end{cases}$$

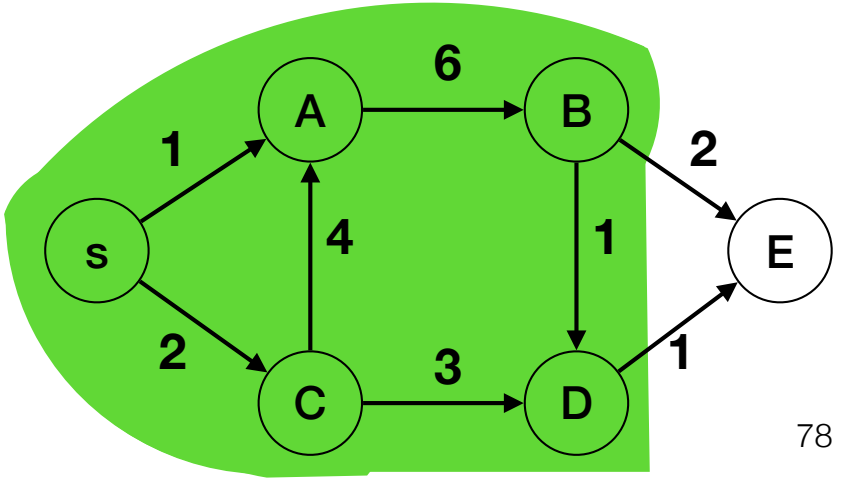
- 问题1：LP_u是LP_v的子问题吗？
- 问题2：子问题的求解顺序如何？

有向无环图中的最长路径

$$OPT(v) = \begin{cases} 0 & \text{if } v = s \\ \max_{u \to v}\{OPT(u) + w(u \to v)\} & \text{otherwise} \end{cases}$$

- 能否据此设计一个求解最长路径的动态规划算法🤯
- 问题1：LP_u是LP_v的子问题吗？
  - $OPT(D) = \max\{OPT(B) + 1, OPT(C) + 3\}$
  - 子问题：在一个较小的有向无环图中求解最长路径
- 问题2：子问题的求解顺序如何？
  - 拓扑排序





动态规划算法的正确性

- 递归关系

$$OPT(v) = \begin{cases} 0 & \text{if } v = s \\ \max_{u \to v}\{OPT(u) + w(u \to v)\} & \text{otherwise} \end{cases}$$

- 上述递归关系对于有向无环图来说是正确的
  - 总是可以归约到基本情况：不存在环！
- 上述递归关系对于无向图来说并不正确
  - 存在无限递归的可能
    - 无向图中如果有环
  - 需要记录一些信息，避免将一个顶点多次加入到路径中
    - 比如，记录当前路径

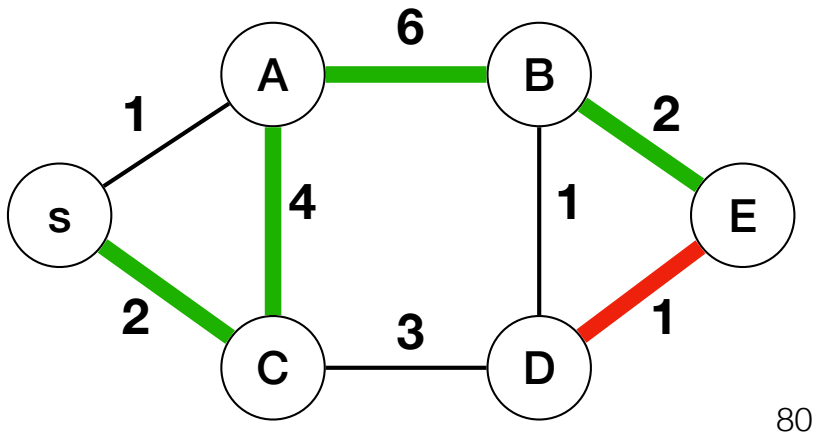
记录当前路径

$$\mathscr{P}(n,k) = \binom{n}{k} k!$$

- 令$OPT(v,P)$是s经路径P到v的最长简单路径的长度

$$OPT(v,P) = \max_{\substack{u \to v,\ v \notin P' \\ P = P' + v}} \{OPT(u,P') + w(u \to v)\}$$

- 原问题即为求解$\max_{P}\{OPT(t,P)\}$
- 不要被max迷惑：$OPT(v,P)$仅仅依赖于一个子问题$OPT(u,P')$
  - $OPT(D, C \to A \to B \to E) = OPT(E, C \to A \to B) + w(E \to D)$
- 子问题的数量如何？ → $(n-1)\sum_{k=0}^{n-2} \mathscr{P}(n-2,k)$
  - $0 \le$ 路径P中的顶点数 $\le n-2$
    - $\sum_{k=0}^{n-2} \mathscr{P}(n-2,k)$
  - $v \in V \backslash \{s\}$，共$n-1$种取值可能

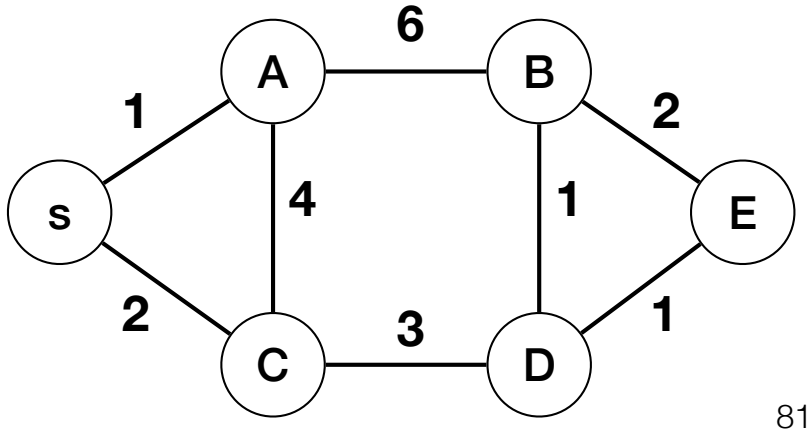


记录当前路径的另一种尝试

- 令$OPT(v, S)$是s经集合S中任意顶点到v的最长简单路径的长度

$$OPT(v, S) = \max \left\{ OPT(v, \varnothing), \max_{\substack{u \to v,\ v \notin S' \\ S = S' \bigcup \{v\}}} \{OPT(u, S') + w(u \to v)\} \right\}$$

- 原问题即为求解$OPT(t, V \backslash \{s, t\})$
- 子问题的数量如何？
  - 集合S中的顶点数为$n-2$
  - $v \in V \backslash \{s\}$，共$n-1$种取值可能
  - $(n-1) \cdot 2^{n-2}$



无向图中最长路径算法的比较

- 记录当前完整路径的动态规划：$(n-1)\sum_{k=0}^{n-2}\mathscr{P}(n-2,k)$
- 记录路径中可能顶点的动态规划：$(n-1)\cdot 2^{n-2}$

两种动态规划算法子问题数量比较

n	DP_1	DP_2
6	325	80
7	1,956	192
8	13,699	448
9	109,600	1,024
10	986,409	2,304
15	236,975,164,804	114,688
20	330,665,665,962,404,030	4,980,736