

# 动态规划

机器学习研究室

计算机科学与技术学院  
吉林大学

# 什么是动态规划？

- 多阶段决策过程 (multistep decision process)

是指这样一类特殊的活动过程，该过程可以按时间顺序分解成若干个相互联系的阶段，在每一个阶段都需要做出决策，全部过程的决策是一个决策序列。

动态规划 (dynamic programming) 算法是解决多阶段决策过程最优化问题的一种技巧性较强的常用方法，。

- 利用动态规划算法，可以优雅而高效地解决很多贪婪算法或分治算法不能解决的问题。

# 什么是动态规划？

- 动态规划算法将问题的解决方案视为一系列决策的结果
- 与贪婪算法不同的是，每采用一次贪婪准则，便做出一个不可撤回的决策；而在动态规划算法中，还要考察每个最优决策序列中是否包含一个最优决策子序列，即问题是否具有最优子结构性质。



# 适合使用动态规划求解的问题

- 动态规划算法的有效性依赖于待求解问题本身具有的两个重要性质：最优子结构性质和子问题重叠性质。
- 1、**最优子结构性质**。如果问题的最优解所包含的子问题的解也是最优的，我们就称该问题具有最优子结构性质（即满足最优化原理）。最优子结构性质为动态规划算法解决问题提供了重要线索。
- 2、**子问题重叠性质**。子问题重叠性质是指在用递归算法自顶向下对问题进行求解时，每次产生的子问题并不总是新问题，有些子问题会被重复计算多次。动态规划算法正是利用了这种子问题的重叠性质，对每一个子问题只计算一次，然后将其计算结果保存在一个表格中，当再次需要计算已经计算过的子问题时，只是在表格中简单地查看一下结果，从而获得较高的解题效率。

# 最优化原理（最优子结构性质）

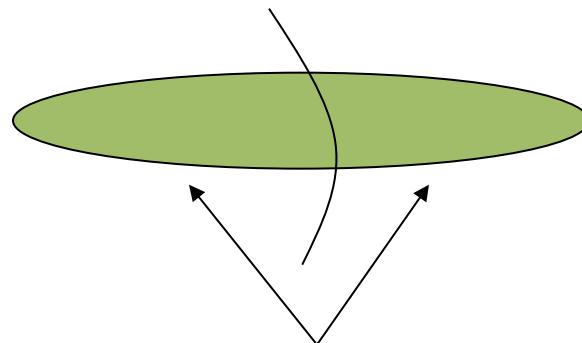
- 作为一个全过程的最优策略具有这样的性质：对于最优策略过程中的任意状态而言，无论其过去的状态和决策如何，余下的诸决策必构成一个最优子策略。即：

一个最优策略的子策略总是最优的
- 一个问题满足最优化原理，又称其具有最优子结构性质。

# 动态规划的两条重要性质

- 一个优化问题能够用动态规划方法求解的关键在于：

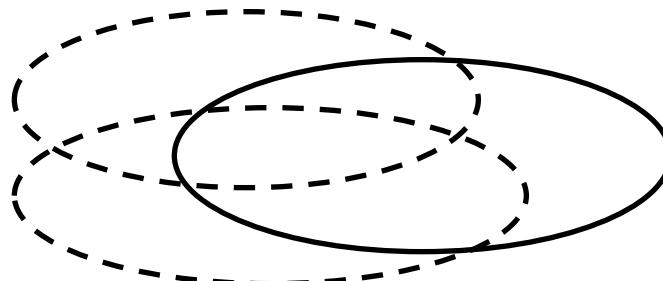
## 1. 最优子结构



每个子结构都是最优的  
(最优化理论)

# 动态规划的两条关键性质

## 2. 重叠子问题



子问题之间不是独立的，  
大量子问题是重叠的

- 动态规划算法正是利用了这种子问题的重叠性质，对每一个子问题只解一次，而后将其解保存在一个表格中，在以后尽可能多地利用这些子问题的解

# 动态规划算法的三个基本元素

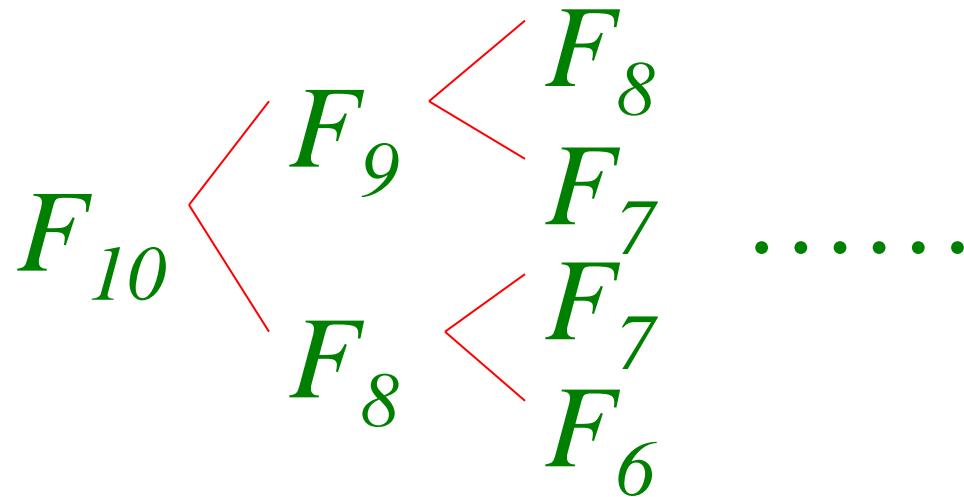
- 递推关系 (recurrence relation)
  - for defining the value of an optimal solution
- 表格计算 (tabular computation)
  - for computing the value of an optimal solution
- 回溯 (traceback)
  - for delivering an optimal solution

# 动态规划算法举例

- 例子1： Fibonacci数
  - 斐波纳契(Fibonacci)数可由如下递归定义：
    - $F(0)=0;$
    - $F(1)=1;$
    - $F(i)=F(i-1)+F(i-2); \text{ For } i>1$

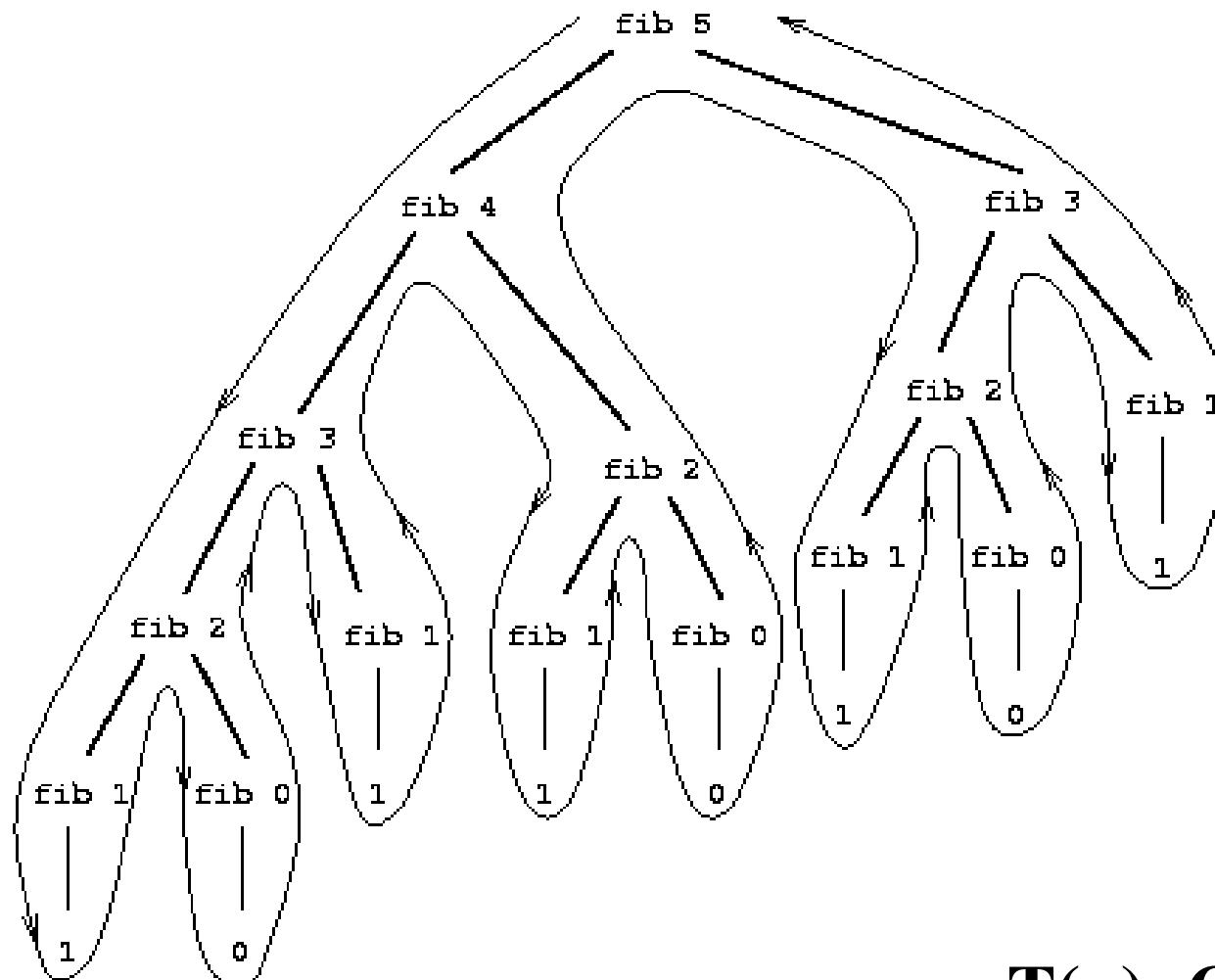
# 动态规划算法举例

- 如何计算F10?



- 递归算法:
  - If  $n=0$ , then  $F=0$ ;
  - Else if  $n=1$ , then  $F=1$ ;
  - Else  $F=F(n-1)+F(n-2)$ ;  
 $T(n)=O(2^n)$

# 动态规划算法举例



-  $T(n)=O(2^n)$

# 动态规划算法

- 表格计算可避免冗余性：用数组将前n-1个数存起来，每次只用一个加法  $F[n] = F[n-1] + F[n-2]$  即可

$F_0$	$F_1$	$F_2$	$F_3$	$F_4$	$F_5$	$F_6$	$F_7$	$F_8$	$F_9$	$F_{10}$
0	1	1	2	3	5	8	13	21	34	55

```
Algorithm F(n);
    integer: T[0..n];
    T[0] = T[1] = 1;
    for i = 2 to n do
        T[i] = T[i-1] + T[i-2];
    F = T[n];
```

**T(n)=O(n)**

# 动态规划算法举例

- 例子2：最大和区间问题
  - 对于一给定的实数序列 $a_1a_2\dots a_n$ ，找出和最大的连续子序列.

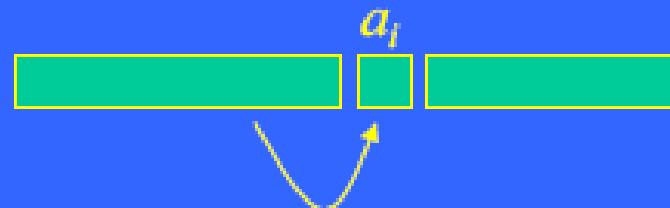
9 -3 1 7 -15 2 3 -4 2 -7 6 -2 8 4 -9

- 对于每个位置, 我们都可以用 $O(n)$ 时间计算由这个位置开始的最大和区间. 这样寻找这个序列的最大和区间的时间复杂度为 $O(n^2)$

# 最大和区间的递归关系

- 定义  $S(i)$  为在  $i$  点结束的最大和区间的值
- 如果  $S(i-1) < 0$ , 则  $a_i$  与其前最大和区间连接将小于它本身, 故有:

$$S(i) \leftarrow a_i + \max \begin{cases} S(i-1) \\ 0 \end{cases}$$



# 最大和区间的表格计算

9 -3 1 7 -15 2 3 -4 2 -7 6 -2 8 4 -9

$S(i)$  9 6 7 14 -1 2 5 1 3 -4 6 4 12 16 7

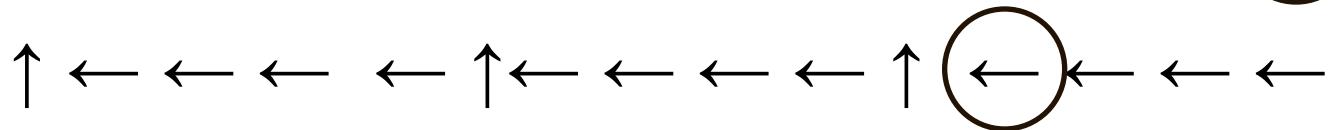
↑ ← ← ← ← ↑ ← ← ← ← ↑ ← ← ←

最大的 $S(i)$ 值

# 最大和区间的回溯过程

9 -3 1 7 -15 2 3 -4 2 -7 6 -2 8 4 -9

$S(i)$  9 6 7 14 -1 2 5 1 3 -4 6 4 12 16 7



**最大和区间为: 6 -2 8 4**

**时间复杂度为:  $O(n)$**

## 例: 最长公共子序列问题

- 子序列: S的子序列是由S删除掉0个或多个字符后得到的.
- 如”president”的子序列可为: pred, sdn, predent等.
- 最长公共子序列问题, 顾名思义, 是确定两个序列之公共子序列中长度最大者.

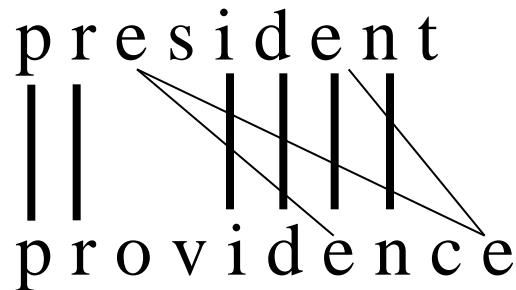
# 最长公共子序列问题

(the longest common subsequence, LCS)

- 举例1

- 序列1: president
- 序列2: providence

最长公共子序列为



# 最长公共子序列的递归关系

- 令  $A=a_1 a_2 \dots a_m$  和  $B=b_1 b_2 \dots b_n$
- 定义  $\text{len}(i, j)$ : 字符串  $a_1 a_2 \dots a_i$  和  $b_1 b_2 \dots b_j$  的最长公共子序列的长度
- 从一个适当的初始化开始,  $\text{len}(i, j)$  可以按照下式计算

$$\text{Len}(i, j) = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0, \\ \text{len}(i-1, j-1)+1 & \text{if } i, j>0 \text{ and } a_i = b_j, \\ \max(\text{len}(i, j-1), \text{len}(i-1, j)) & \text{if } i, j>0 \text{ and } a_i \neq b_j, \end{cases}$$

# 最长公共子序列的递归计算

```
procedure LCS-Length( $A, B$ )
    1. for  $i \leftarrow 0$  to  $m$  do  $len(i, 0) = 0$ 
    2. for  $j \leftarrow 1$  to  $n$  do  $len(0, j) = 0$ 
    3. for  $i \leftarrow 1$  to  $m$  do
        4.     for  $j \leftarrow 1$  to  $n$  do
            5.         if  $a_i = b_j$  then  $\begin{cases} len(i, j) = len(i - 1, j - 1) + 1 \\ prev(i, j) = "↖" \end{cases}$ 
            6.             else if  $len(i - 1, j) \geq len(i, j - 1)$ 
                7.                 then  $\begin{cases} len(i, j) = len(i - 1, j) \\ prev(i, j) = "↑" \end{cases}$ 
                8.                 else  $\begin{cases} len(i, j) = len(i, j - 1) \\ prev(i, j) = "←" \end{cases}$ 
            9.     return  $len$  and  $prev$ 
```

# 最长公共子序列的填表计算

i \ j	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1 p	0	1 ↙	1 ↙	1 ↙	1 ↙	1 ↙	1 ↙	1 ↙	1 ↙	1 ↙	1 ↙
2 r	0	1 ↗	2 ↗	2 ↗	2 ↗	2 ↗	2 ↗	2 ↗	2 ↗	2 ↗	2 ↗
3 e	0	1 ↗	2 ↗	2 ↗	2 ↗	2 ↗	2 ↗	3 ↗	3 ↗	3 ↗	3 ↗
4 s	0	1 ↗	2 ↗	2 ↗	2 ↗	2 ↗	2 ↗	3 ↗	3 ↗	3 ↗	3 ↗
5 i	0	1 ↗	2 ↗	2 ↗	2 ↗	3 ↗	3 ↗	3 ↗	3 ↗	3 ↗	3 ↗
6 d	0	1 ↗	2 ↗	2 ↗	2 ↗	3 ↗	4 ↗	4 ↗	4 ↗	4 ↗	4 ↗
7 e	0	1 ↗	2 ↗	2 ↗	2 ↗	3 ↗	4 ↗	5 ↗	5 ↗	5 ↗	5 ↗
8 n	0	1 ↗	2 ↗	2 ↗	2 ↗	3 ↗	4 ↗	5 ↗	6 ↗	6 ↗	6 ↗
9 t	0	1 ↗	2 ↗	2 ↗	2 ↗	3 ↗	4 ↗	5 ↗	6 ↗	6 ↗	6 ↗

Running time and memory:  $O(mn)$  and  $O(mn)$ .

# 最长公共子序列的回溯过程

## The backtracing algorithm

```
procedure Output-LCS( $A$ ,  $prev$ ,  $i$ ,  $j$ )
1  if  $i = 0$  or  $j = 0$  then return
2  if  $prev(i, j) = "↖"$  then  $\begin{cases} Output - LCS(A, prev, i-1, j-1) \\ \text{print } a_i \end{cases}$ 
3  else if  $prev(i, j) = "↑"$  then  $Output - LCS(A, prev, i-1, j)$ 
4  else  $Output - LCS(A, prev, i, j-1)$ 
```

# 最长公共子序列的回溯过程

i	j	0	1	2	3	4	5	6	7	8	9	10
0		0	0	0	0	0	0	0	0	0	0	0
1	p	0	1	1	1	1	1	1	1	1	1	1
2	r	0	1	2	2	2	2	2	2	2	2	2
3	e	0	1	2	2	2	2	2	3	3	3	3
4	s	0	1	2	2	2	2	2	3	3	3	3
5	i	0	1	2	2	2	3	3	3	3	3	3
6	d	0	1	2	2	2	3	4	4	4	4	4
7	e	0	1	2	2	2	3	4	5	5	5	5
8	n	0	1	2	2	2	3	4	5	6	6	6
9	t	0	1	2	2	2	3	4	5	6	6	6

输出为: priden

# 生物信息学中的序列比对问题

Sequence A: CTTAACT

Sequence B: CGGATCAT

An alignment of A and B:

C---TTAACT ← Sequence A  
CGGATCA--T ← Sequence B

# 生物信息学中的序列比对问题

Sequence A: CTTAACT

Sequence B: CGGATCAT

An alignment of A and B:

Match

Mismatch

C	- - -	T	T	A	A	C
CGGA	T	C	A	- - T		

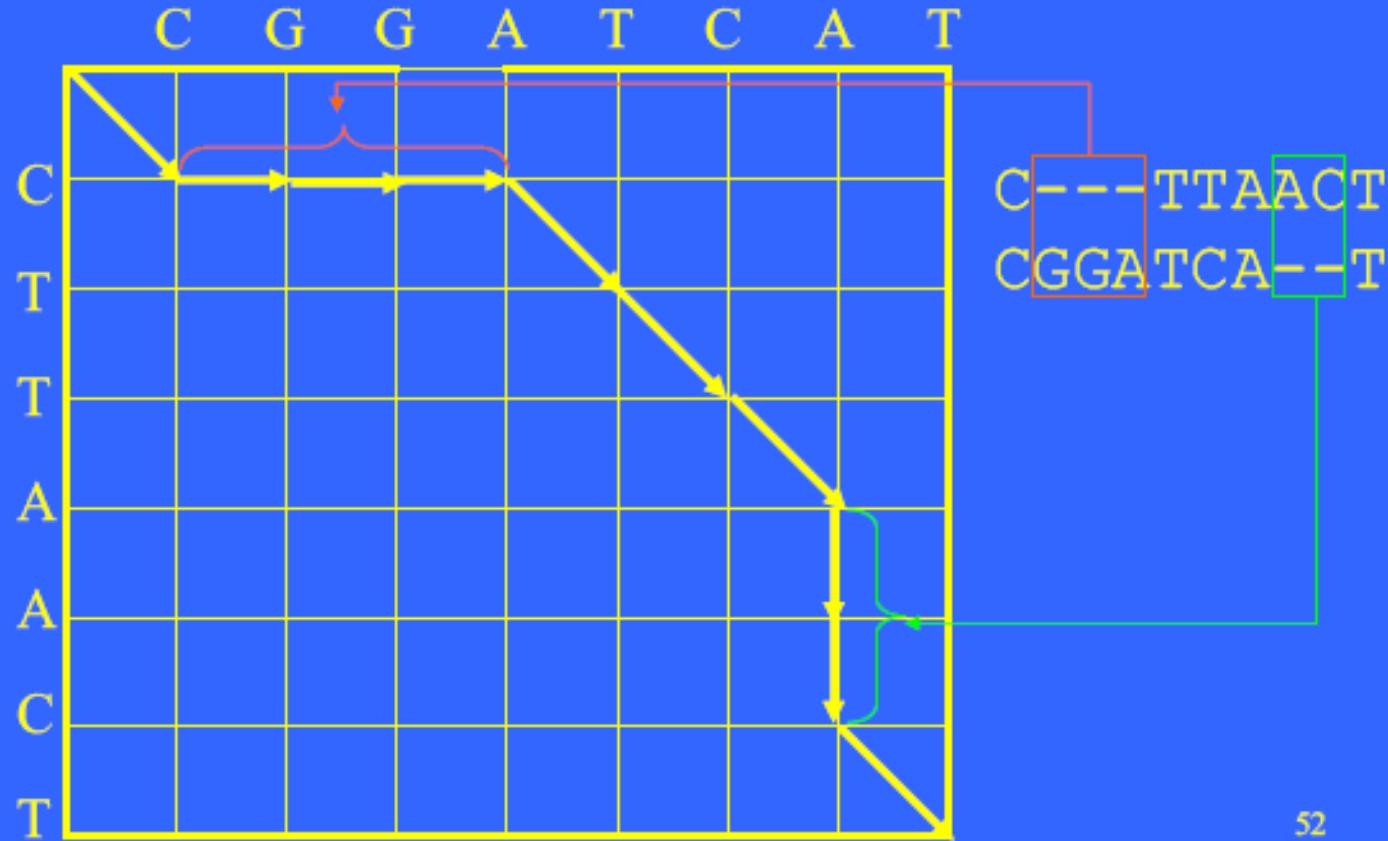
Insertion  
gap

Deletion  
gap

# 序列比对图

Sequence A: CTTAACT

Sequence B: CGGATCAT



52

# 一个简单的得分方案

## A simple scoring scheme

- Match: +8 ( $w(x, y) = 8$ , if  $x = y$ )
- Mismatch: -5 ( $w(x, y) = -5$ , if  $x \neq y$ )
- Each gap symbol: -3 ( $w(-, x) = w(x, -) = -3$ )  
(i.e. space)

C	-	-	-	T	T	A	A	C	T
C	G	G	A	T	C	A	-	-	T
+8	-3	-3	-3	+8	-5	+8	-3	-3	+8

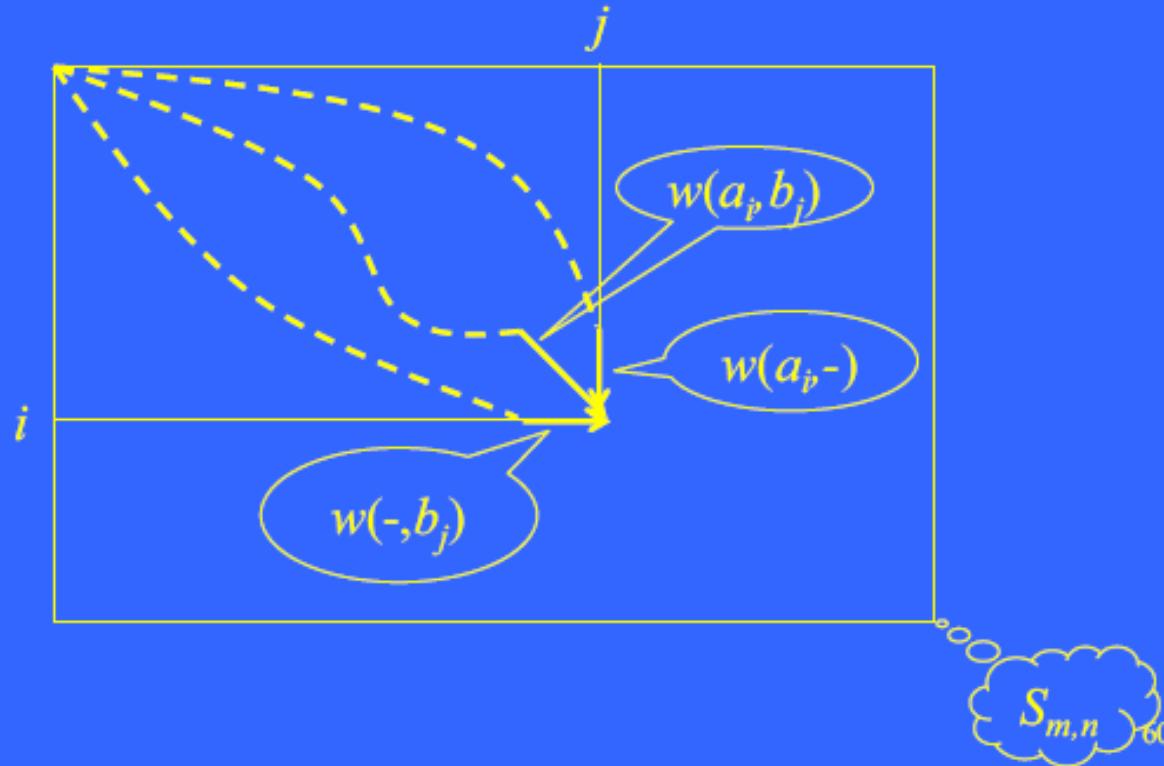
$$= +12$$

alignment score

53

# 分值的递归计算

Computing  $S_{i,j}$



# 分值的递归计算

- 令  $A = a_1 \ a_2 \ \dots \ a_m$  和  $B = b_1 \ b_2 \ \dots \ b_n$
- 定义  $S_{i,j}$ : 字符串  $a_1 \ a_2 \ \dots \ a_i$  和  $b_1 \ b_2 \ \dots \ b_j$  的最优比对(alignment)得分
- 从一个适当的初始化开始，  $S_{i,j}$  可以按照下式计算

$$S_{i,j} = \max \begin{cases} S_{i-1, j} + w(a_i, -) \\ S_{i-1, j-1} + w(a_i, b_j) \\ S_{i, j-1} + w(-, b_j) \end{cases}$$

# 填表计算

## Initialization

	C	G	G	A	T	C	A	T	
C	0	-3	-6	-9	-12	-15	-18	-21	-24
T	-3								
T	-6								
T	-9								
A	-12								
A	-15								
C	-18								
T	-21								

# 填表计算

$$S_{3,5} = ?$$

	C	G	G	A	T	C	A	T	
C	0	-3	-6	-9	-12	-15	-18	-21	-24
T	-3	8	5	2	-1	-4	-7	-10	-13
T	-6	5	3	0	-3	7	4	1	-2
T	-9	2	0	-2	-5	?			
A	-12								
A	-15								
C	-18								
T	-21								

# 填表计算

$$S_{3,5} = ?$$

	C	G	G	A	T	C	A	T	
C	0	-3	-6	-9	-12	-15	-18	-21	-24
T	-3	8	5	2	-1	-4	-7	-10	-13
T	-6	5	3	0	-3	7	4	1	-2
T	-9	2	0	-2	-5	5	-1	-4	9
A	-12	-1	-3	-5	6	3	0	7	6
A	-15	-4	-6	-8	3	1	-2	8	5
C	-18	-7	-9	-11	0	-2	9	6	3
T	-21	-10	-12	-14	-3	8	6	4	14

optimal  
score

63

# 填表计算

C	T	T	A	A	C	-	T	
C	G	G	A	T	C	A	T	
8	-5	-5	+8	-5	+8	-3	+8	= 14

	C	G	G	A	T	C	A	T	
C	0	-3	-6	-9	-12	-15	-18	-21	-24
T	-3	8	5	2	-1	-4	-7	-10	-13
T	-6	5	3	0	-3	7	4	1	-2
T	-9	2	0	-2	-5	5	-1	-4	9
A	-12	-1	-3	-5	6	3	0	7	6
A	-15	-4	-6	-8	3	1	-2	8	5
C	-18	-7	-9	-11	0	-2	9	6	3
T	-21	-10	-12	-14	-3	8	6	4	14

# 局部序列比对

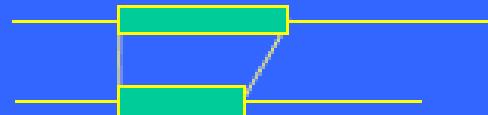
局部序列比对就是对于两条给定序列, 寻找它们之间得分最大的子序列比对

## Global Alignment vs. Local Alignment

- global alignment:



- local alignment:



# 局部序列比对

## An optimal local alignment

- $S_{i,j}$ : the score of an optimal local alignment ending at  $a_i$  and  $b_j$
- With proper initializations,  $S_{i,j}$  can be computed as follows.

$$S_{i,j} = \max \begin{cases} 0 \\ S_{i-1,j} + w(a_i, -) \\ S_{i,j-1} + w(-, b_j) \\ S_{i-1,j-1} + w(a_i, b_j) \end{cases}$$

# 局部序列比对

Match: 3

Mismatch: -5

Gap symbol: -3

local alignment

	C	G	G	A	T	C	A	T	
C	0	8	5	2	0	0	8	5	2
T	0	5	3	0	0	8	5	3	13
T	0	2	0	0	0	8	5	2	11
A	0	0	0	0	8	5	3	?	
A	0								
C	0								
T	0								

# 局部序列比对

Match: 8

Mismatch: -5

Gap symbol: -3

local alignment

		C	G	G	A	T	C	A	T
		0	0	0	0	0	0	0	0
C	0	8	5	2	0	0	8	5	2
	T	0	5	3	0	0	8	5	3
	T	0	2	0	0	0	8	5	2
	A	0	0	0	0	8	5	3	11
	A	0	0	0	0	8	5	2	11
	C	0	8	5	2	5	3	13	10
	T	0	5	3	0	2	13	10	8
	T	0	5	3	0	2	13	10	18



68

# 局部序列比对

A - C - T  
A T C A T  
 $8-3+8-3+8 = 18$

	C	G	G	A	T	C	A	T	
C	0	8	5	2	0	0	8	5	2
T	0	5	3	0	0	8	5	3	13
T	0	2	0	0	0	8	5	2	11
A	0	0	0	0	8	5	3	13	10
A	0	0	0	0	8	5	2	11	8
C	0	8	5	2	5	3	13	10	7
T	0	5	3	0	2	13	10	8	18*

(not always at  
the corner)



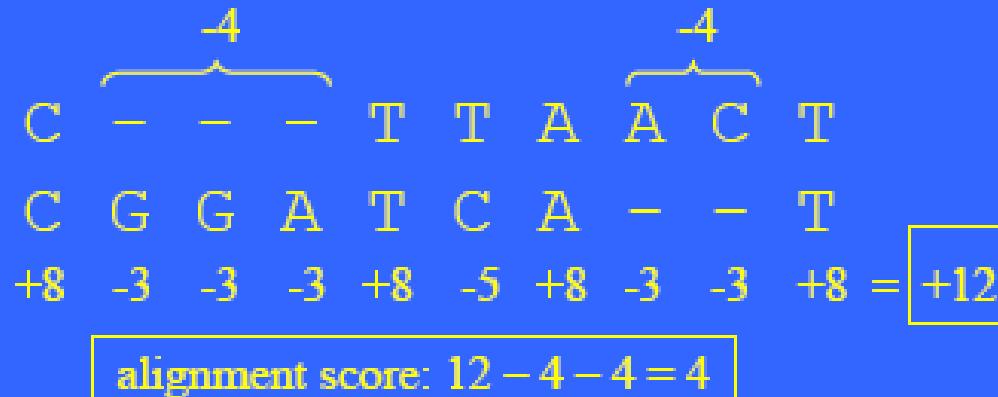
69

# 带窗口罚分的序列比对

带开窗口罚分的序列比对即在正常的序列比对问题中除了考虑每个空格的罚分, 还考虑了开空格窗口的罚分.

## Affine gap penalties

- Match: +8 ( $w(x, y) = 8$ , if  $x = y$ )
- Mismatch: -5 ( $w(x, y) = -5$ , if  $x \neq y$ )
- Each gap symbol: -3 ( $w(-, x) = w(x, -) = -3$ )
- E.g. each gap is charged an extra gap-open penalty: -4.
- In general, a gap of length  $k$  should have penalty  $g(k)$



# 带窗口罚分的序列比对

## Affine gap penalties

- A gap of length  $k$  is penalized  $x + k \cdot y$ .

↑  
gap-open penalty  
↑

Three cases for alignment endings:

gap-symbol penalty

1.  $\dots x \}$  an aligned pair  
 $\dots x \}$

2.  $\dots x \}$  a deletion  
 $\dots - \}$

3.  $\dots - \}$  an insertion  
 $\dots x \}$

# 带窗口罚分的序列比对

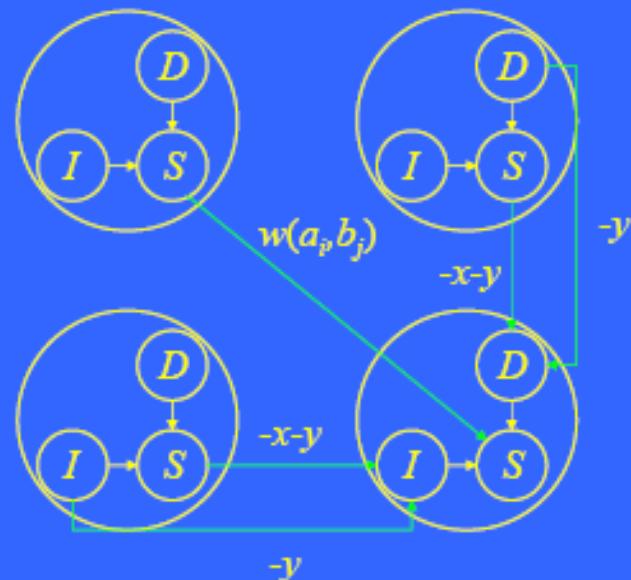
## Affine gap penalties

- Let  $D(i, j)$  denote the maximum score of any alignment between  $a_1a_2\dots a_i$  and  $b_1b_2\dots b_j$  ending with a deletion.
- Let  $I(i, j)$  denote the maximum score of any alignment between  $a_1a_2\dots a_i$  and  $b_1b_2\dots b_j$  ending with an insertion.
- Let  $S(i, j)$  denote the maximum score of any alignment between  $a_1a_2\dots a_i$  and  $b_1b_2\dots b_j$ .

# 带窗口罚分的序列比对

$$D(i, j) = \max \begin{cases} D(i-1, j) - y \\ S(i-1, j) - x - y \end{cases}$$
$$I(i, j) = \max \begin{cases} I(i, j-1) - y \\ S(i, j-1) - x - y \end{cases}$$
$$S(i, j) = \max \begin{cases} S(i-1, j-1) + w(a_i, b_j) \\ D(i, j) \\ I(i, j) \end{cases}$$

Affine gap penalties  
(Gotoh's algorithm)

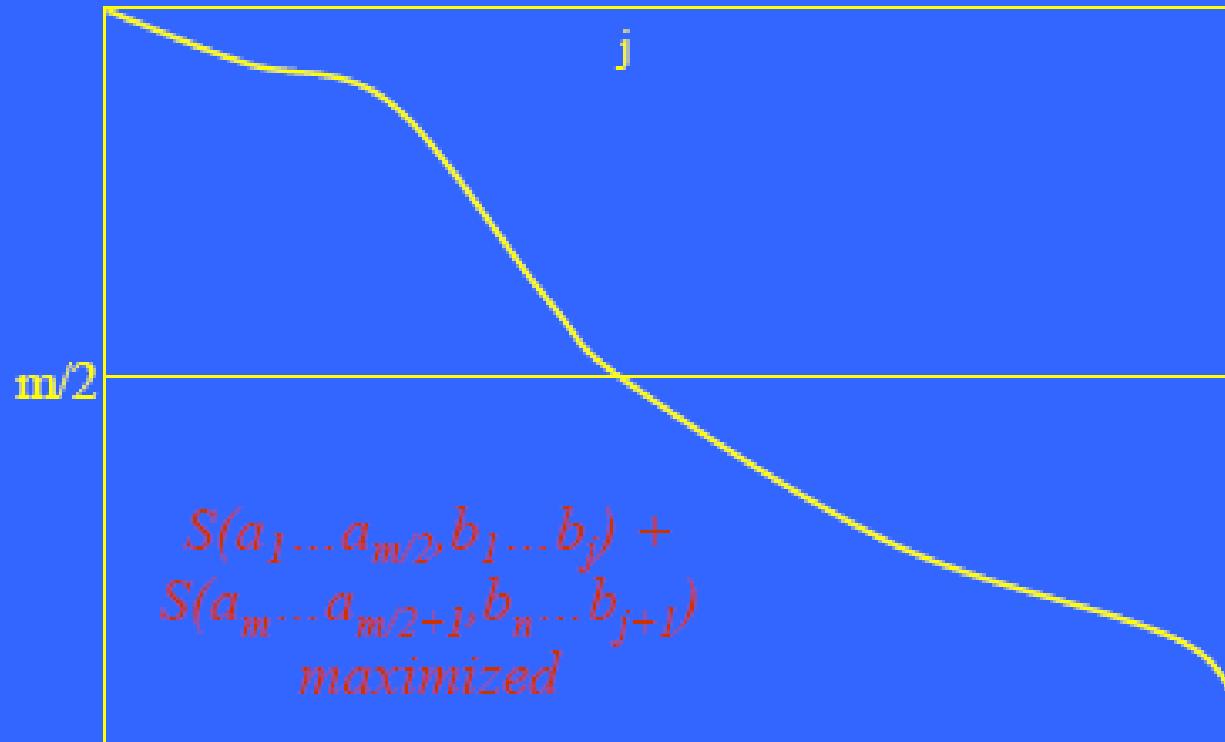


# 线性空间的方法

## Linear space ideas

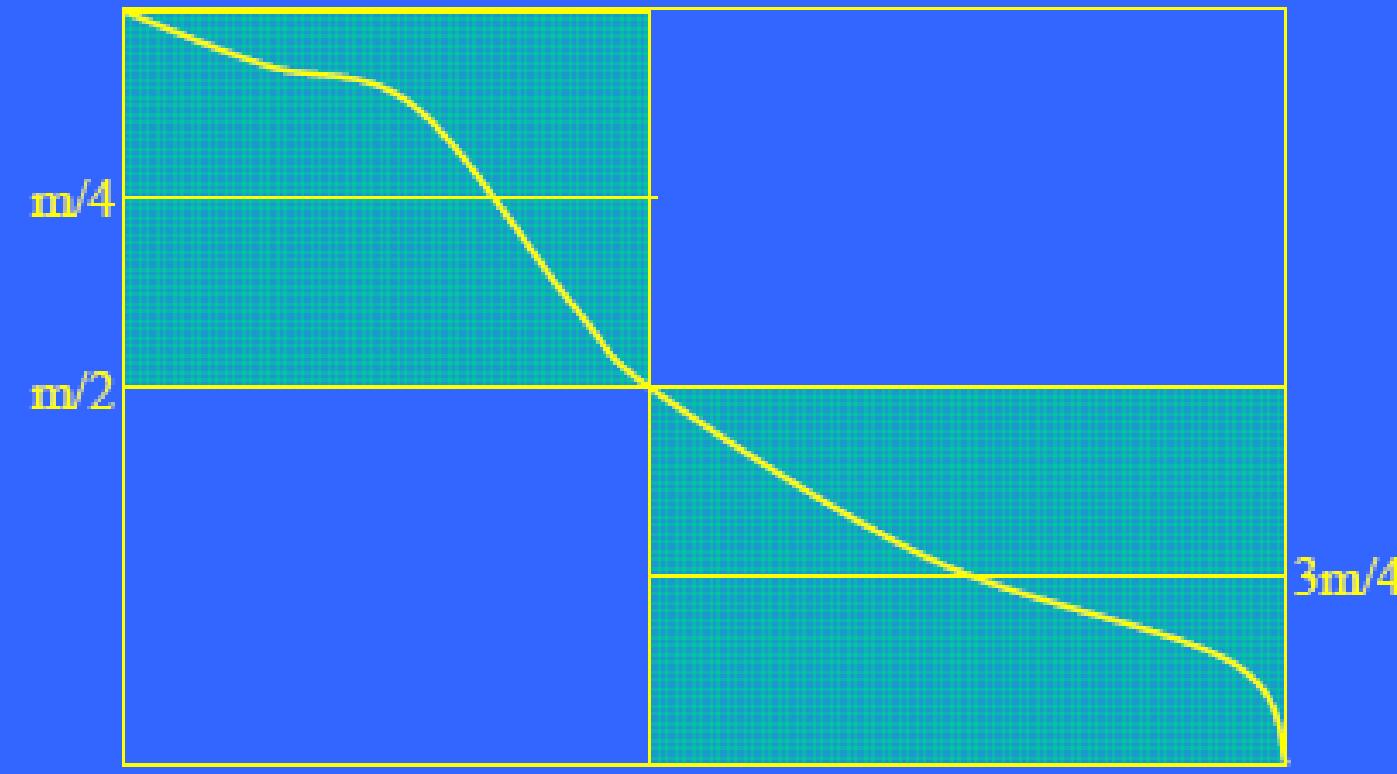
Hirschberg, 1975; Myers and Miller, 1988

- (i) scores can be computed in  $O(n)$  space
- (ii) divide-and-conquer



# 线性空间的方法

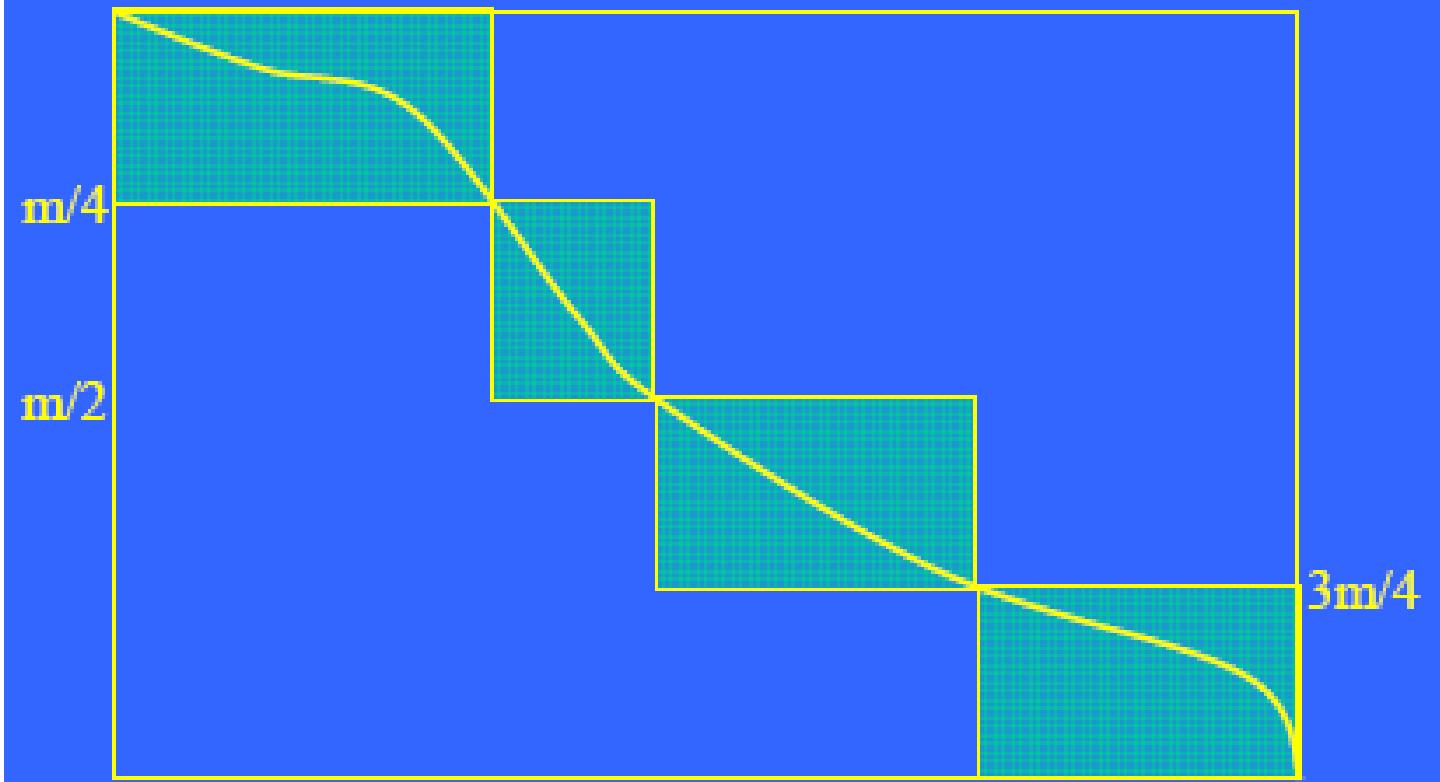
Two subproblems  
 $\frac{1}{2}$  original problem size



# 线性空间的方法

Four subproblems

$\frac{1}{4}$  original problem size



# 线性空间的方法

## Time and Space Complexity

- Space:  $O(m+n)$
- Time:

$$O(mn) * \underbrace{\left(1 + \frac{1}{2} + \frac{1}{4} + \dots\right)}_2 = O(mn)$$