PKU-04833060
算法设计与分析『实验班』

第4讲 动态规划 (1/2)

罗国杰

gluo@pku.edu.cn

2024年春季学期

计算 Fibonacci 数

Fibonacci 数

1, 1, 2, 3, 5, 8, 13, 21, …

满足 $F_n=F_{n-1}+F_{n-2}$

增加$F_0=0$，得到数列　0, 1, 1, 2, 3, 5, 8, 13, 21, …

通常算法：从 $F_0, F_1, \ldots$, 根据定义陆续相加，时间为$\Theta(n)$

定理2.1 设 $\{F_n\}$为 Fibonacci 数构成的数列，那么

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

算法：令矩阵 $M=\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$，用分治法计算 M^n

时间 $T(n)=\Theta(\log n)$.

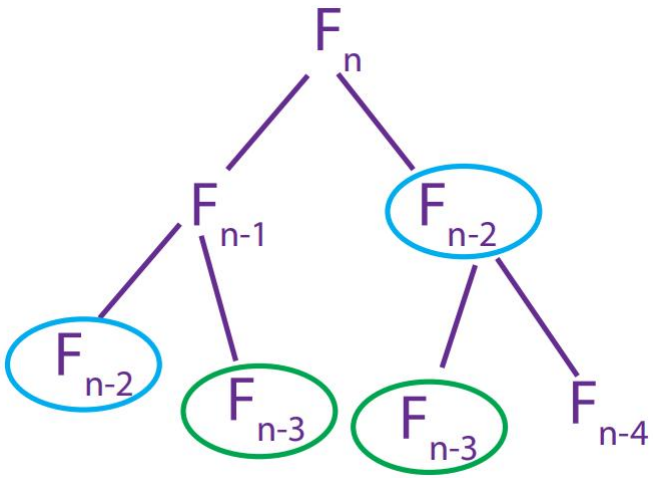
第n个斐波那契数

- 第n个斐波那契数
  - $F_1 = F_2 = 1$; $F_n = F_{n-1} + F_{n-2}$
  - 计算 F_n 的值

- 直接实现递归定义

```
fib(n):
    if n ≤ 2: return 1
    else: return fib(n-1)+fib(n-2)
```

  - $T(n) = T(n-1) + T(n-2) + O(1) \geq F_n$
  - $T(n) \geq 2\ T(n-2) + O(1) \geq 2^{n/2}$



第n个斐波那契数

- 第n个斐波那契数
  - $F_1 = F_2 = 1$; $F_n = F_{n-1} + F_{n-2}$
  - 计算 F_n 的值

- 直接实现递归定义

```
fib(n):
    if n ≤ 2: return 1
    else: return fib(n-1)+fib(n-2)
```

  - $T(n) = T(n-1) + T(n-2) + O(1) \geq F_n$
  - $T(n) \geq 2\ T(n-2) + O(1) \geq 2^{n/2}$

- 记忆化的动态规划

```
memo = {}
fib(n):
    if n in memo: return memo[n]
    else: if n ≤ 2: f = 1
          else: f = fib(n-1)+fib(n-2)
          memo[n] = f
          return f
```

  - 命中 memo 的调用是 O(1) 时间
  - 最多产生 n 次 memo 缺失的情况
  - 总共 O(n) 时间

Memoization using C++ (though ugly)

```
#include <functional>
#include <iostream>
#include <tuple>
#include <map>

template <typename ReturnType, typename... Args>
std::function<ReturnType (Args...)> memorize(
    std::function<ReturnType (Args...)> func) {
  std::map<std::tuple<Args...>, ReturnType> cache;
  return ([=](Args... args) mutable {
    std::tuple<Args...> t(args...);
    if (cache.find(t) == cache.end())
    cache[t] = func(args...);
    return cache[t];
  });
}
```

递归版本

```
int fib0(int n) {
  if  (n < 2) return n;
  return fib0(n-1) + fib0(n-2);
}
```

记忆化版本

```
std::function<int (int)> fib;
struct MemorizedFib {
  static int f(int n) {
    if  (n < 2) return n;
    return fib(n-1) + fib(n-2);
  }
  MemorizedFib() {
    fib = memorize(std::function<int (int)>(f));
  }
} mfib;
```

```
int main(void) {
  int n = 45;

  std::cout << "memorized version: fib(" << n << ") = "
            << fib(n) << std::endl << std::flush;

  std::cout << "original version: fib0(" << n << ") = "
            << fib0(n) << std::endl << std::flush;
}
```

第n个斐波那契数

- 第n个斐波那契数
  - $F_1 = F_2 = 1$; $F_n = F_{n-1} + F_{n-2}$
  - 计算 F_n 的值

- 直接实现递归定义

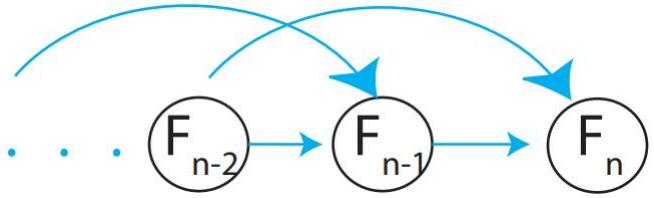
```
fib(n):
    if n ≤ 2: return 1
    else: return fib(n-1)+fib(n-2)
```

  - $T(n) = T(n-1) + T(n-2) + O(1) \geq F_n$
  - $T(n) \geq 2\ T(n-2) + O(1) \geq 2^{n/2}$

- 自底向上的动态规划

```
fib = {}
for k in [1, 2, ..., n]:
    if k ≤ 2: f = 1
    else: f = fib[k-1] + fib[k-2]
    fib[k] = f
return fib[n]
```

  - 计算量同记忆化的动规
  - 可看作将递归调用图（返回边）展开后，合并子问题，并按拓扑序遍历



  - 更容易分析和优化：O(n) 时间、O(1) 空间

动态规划：似曾相识的例子

- Bellman-Ford 算法
  - $d_s^{(k)}(v) = \begin{cases} w(s,v) & k = 1 \\ \min\limits_u \{ d_s^{(k-1)}(u) + w(u,v) \} & k > 1 \end{cases}$
    - 若 $u = v$，记 $w(u,v) = 0$
    - 若 $(u,v) \notin E$，记 $w(s,v) = +\infty$
  - 表示从 s 到 v，经过不超过 k 条边的最短路径长度

- Floyd 算法
  - $d^{(k)}(u,v) = \begin{cases} w(u,v) & k = 0 \\ \min \{ d^{(k-1)}(u,v), \quad d^{(k-1)}(u,k) + d^{(k-1)}(k,v) \} & k > 0 \end{cases}$
    - 节点标号 u, v ∈ { 1, 2, …, N }
  - 表示从 u 到 v，内部节点标号不超过 k 的最短路径长度

动态规划（Dynamic Programming）

- 术语“dynamic programming”由美国应用数学家 Richard Bellman (1920-1984) 创造

- 强大而简洁的算法设计技术和运筹学方法
- 一类特殊的优化算法（最小化/最大化）
- 能用多项式时间求解一大类可行解空间是指数规模的问题

- 动规 ≈ 受控的暴力
- 动规 ≈ 递归 + 复用

内容提要

- 若干动态规划例子
- 动态规划的特征
  - 最优子结构
  - 重叠子问题
- 递推的步骤

问题：矩阵链乘法

- 设$A_1, A_2, \ldots, A_n$为矩阵序列，其中A_i为$P_{i-1} \times P_i$阶矩阵，$i=1, 2, \ldots, n$。确定乘法顺序使元素相乘的总次数最少。
- 输入：向量$P = <P_0, P_1, \ldots, P_n>$
- 实例：$P = <10, 100, 5, 50>$

$$A_1: 10 \times 100, A_2: 100 \times 5, A_3: 5 \times 50$$

- 乘法次序

$$(A_1A_2)A_3: 10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$$

$$A_1(A_2A_3): 10 \times 100 \times 50 + 100 \times 5 \times 50 = 75000$$

矩阵链乘法：搜索空间的规模

- n 是乘号的个数
- 搜索空间大小的递推方程：$C_0 = 1$ 和 $C_{n+1} = \sum_{i=0}^{n} C_i \, C_{n-i}$
- 生成函数：$c(x) = \sum_{n=0}^{\infty} C_n x^n \quad \Rightarrow \quad c(x) = 1 + x \cdot c^2(x)$
- 求解得：$c(x) = \frac{1+\sqrt{1-4x}}{2x}$ 或 $c(x) = \frac{1-\sqrt{1-4x}}{2x}$
- 由 $C_0 = \lim_{x \to 0} c(x) = 1$ 得 $c(x) = \frac{1-\sqrt{1-4x}}{2x}$
- 代数展开 $c(x) = \frac{1-\sqrt{1-4x}}{2x} = \sum_{n=0}^{\infty} C_n x^n$ 得 $C_n = \frac{1}{n+1} \cdot \binom{2n}{n}$ （过程略）

矩阵链乘法：递归算法

```
RecurMatrixChain(p,i,j)
1. (minK, minCost) ← (NaN, +∞)
2. for k ← i to j-1 do // 枚举当前问题最后一次乘法位置
3.     q ← RecurMatrixChain(p,i,k)._2
           + RecurMatrixChain(p,k+1,j)._2 + p_{i-1}p_kp_j
4.     if q < minCost
5.     then (minK, minCost) ← (k, q)
6. return (k, minCost)
```

矩阵链乘法：递归算法运行时间

- 运行时间满足递推关系

$$T(n) \geq \begin{cases} \Theta(1) & n = 1 \\ \sum_{k=1}^{n-1} [T(k) + T(n-k) + \Theta(1)] & n > 1 \end{cases}$$

$$T(n) \geq \Theta(n) + \sum_{k=1}^{n-1} T(k) + \sum_{k=1}^{n-1} T(n-k)$$

$$= \Theta(n) + 2\sum_{k=1}^{n-1} T(k), \quad n > 1$$

矩阵链乘法：递归算法运行时间

- 猜测$T(n)=\Omega(2^n)$
- 只需用数学归纳法证明$T(n)\geq 2^{n-1}$
- 当$n=1$时，显然能成立
- 假设对于任何小于n的k命题都成立，则

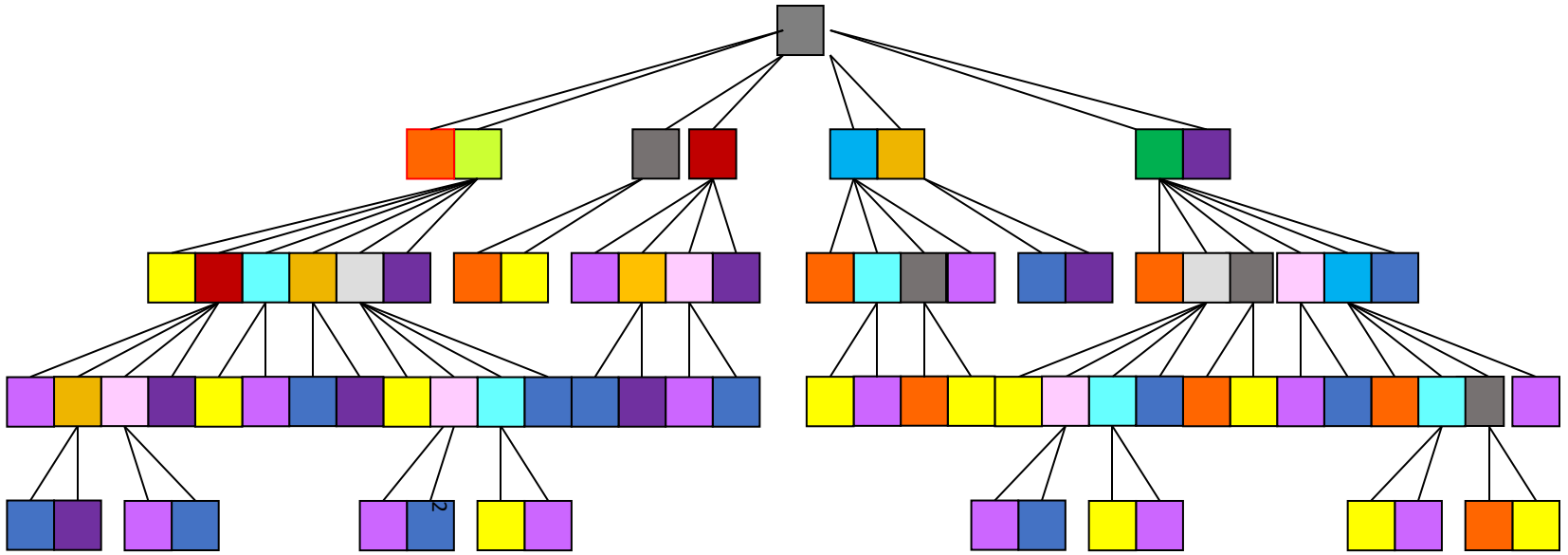
$$T(n)\geq \Theta(n)+2\sum_{k=1}^{n-1}T(k)\geq \Theta(n)+2\sum_{k=1}^{n-1}2^{k-1}$$

$$=\Theta(n)+2\left(2^{n-1}-1\right)\geq 2^{n-1}$$

矩阵链乘法：复杂性高的原因--子问题重复计算

n=5，计算子问题：81个；不同的子问题：15个

子问题	1-1	2-2	3-3	4-4	5-5	1-2	2-3	3-4	4-5	1-3	2-4	3-5	1-4	2-5	1-5
数	8	12	14	12	8	4	5	5	4	2	2	2	1	1	1



矩阵链乘法：递推方程

- 输入为$P=<P_0, P_1, \dots, P_n>$，$A_{i..j}$ 表示乘积$A_iA_{i+1}..A_j$的结果，其最后一次相乘是$A_{i..j}= A_{i..k} A_{k+1..j}$
- $m[i,j]$ 表示得到 $A_{i..j}$的最少的相乘次数，则
- 递推方程

$$m[i,j] = \begin{cases} 0 & i = j \\ \min\{m[i,k] + m[k+1,j] + P_{i-1}P_kP_j\} & i < j \end{cases}$$

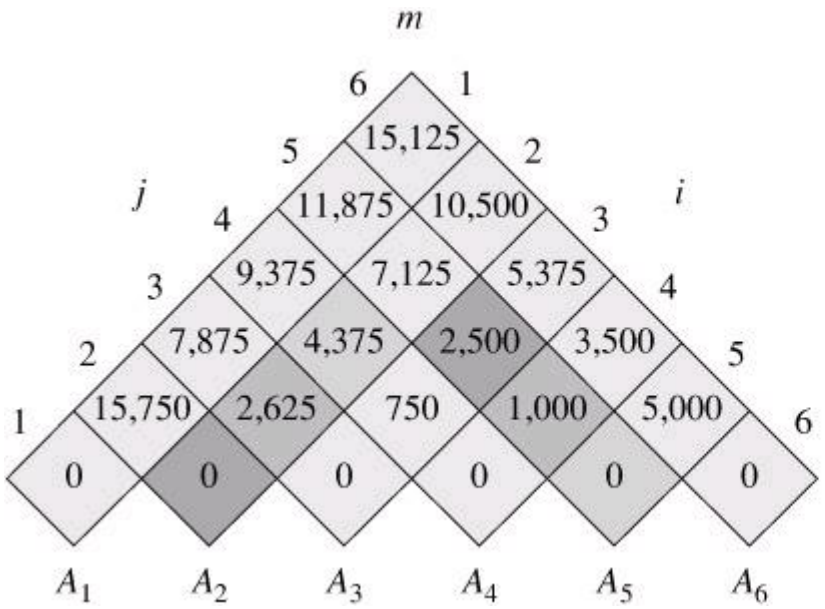
- “标记函数”：设计表$s[i,j]$，记录求得$m[i,j]$最优时最后一次运算的位置
- 正确性：对矩阵链长度 (j-i) 做数学归纳法证明

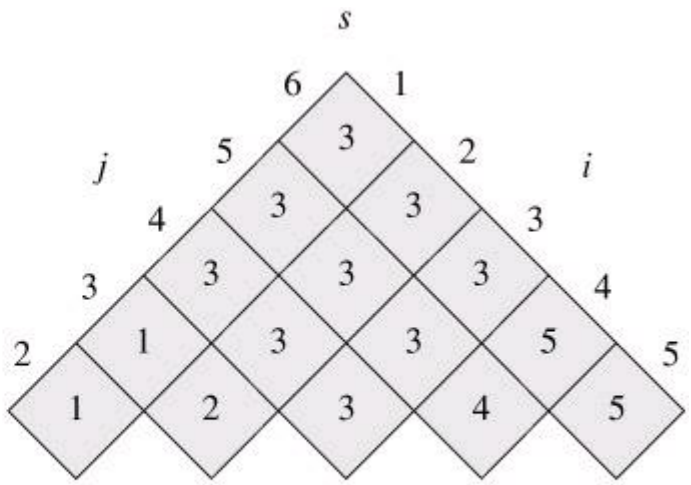
算法：非递归算法

```
MATRIX-CHAIN-ORDER(p)
1  n ← length[p] - 1
2  for i ← 1 to n
3      do m[i, i] ← 0
4  for l ← 2 to n                ▷ l 是子矩阵链长度.
5      do for i ← 1 to n - l + 1
6             do j ← i + l - 1
7                m[i, j] ← ∞
8                for k ← i to j - 1
9                    do q ← m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j
10                      if q < m[i, j]
11                         then m[i, j] ← q
12                              s[i, j] ← k
13 return m and s
```

$T(n)=O(n^3)$

矩阵链乘法：动态规划例子





A_1 30 × 35
A_2 35 × 15
A_3 15 × 5
A_4 5 × 10
A_5 10 × 20
A_6 20 × 25

最优加括号的情形

$((A_1 (A_2 A_3)) ((A_4 A_5)A_6))$

动态规划的特征1

最优子结构
问题的最优解蕴含着
子问题的最优解

矩阵链乘法问题：如果矩阵链 $A_{1,N}$ 最优乘法顺序的最后三次运算是 $(A_{1,L}A_{L+1,M})(A_{M+1,R}A_{R+1,N})$，则对子链 $A_{1,M}$ 和 $A_{M+1,N}$，也分别存在最优解使 $A_{1,L}A_{L+1,M}$ 和 $A_{1,L}A_{L+1,M}$ 分别是各自最后一次乘法。

最优化原则

Bellman's Principle of Optimality

An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision. [Bellman 1957]

动态规划的特征2

重叠子问题

递归中求解的为数不多的子问题被多次重复的计算

矩阵链 $A_{1,N}$ 最优乘法顺序的子问题

仅有 N^2 个，vs. $\frac{1}{N+1}\binom{2N}{N}$

动态规划算法设计步骤

- 将问题表示成多步判断，确定子问题的边界
  - 整个判断序列就对应问题的最优解
  - 每步判断对应一个子问题，子问题类型与原问题一样
- 确定优化函数，以函数的极大（或极小）作为判断的依据
- 确定是否满足优化原则——动态规划算法的必要条件
- 确定子问题的重叠性
- 列出关于优化函数的递推方程（或不等式）和边界条件
  - 正确性的证明
- 自底向上计算子问题的优化函数值
- 备忘录方法（表格）存储中间结果
- 设立标记函数 s[i,j] 求解问题的解

最大子段和

问题：给定n个整数（可以为负数）的序列 $(a_1, a_2, \dots, a_n)$

求 $\max\{\ 0, \max\limits_{1\le i\le j\le n} \sum\limits_{k=i}^{j} a_k\ \}$

实例：(-2, 11, -4, 13, -5, -2)

解：最大子段和 $a_2+a_3+a_4= 20$

算法1 --- 顺序求和+比较

算法2 --- 分治策略

算法3 --- 动态规划

最大子段和 算法1：顺序求和+比较

算法 Enumerate

输入：数组*A*[1..*n*]

输出：*sum, first, last*

```
1.  sum ← 0
2.  for  i ← 1  to  n  do      // i为当前和的首位置
3.    for  j ← i  to  n  do    // j为当前和的末位置
4.      thissum ← 0    // thissum为A[i]到A[j]之和
5.      for  k ← i  to  j  do
6.        thissum ← thissum+A[k]
7.      if  thissum > sum
8.      then  sum ← thissum
9.            first ← i        // 记录最大和的首位置
10.           last ← j         // 记录最大和的末位置
```

时间复杂度：$O(n^3)$

最大子段和 算法2：分治策略

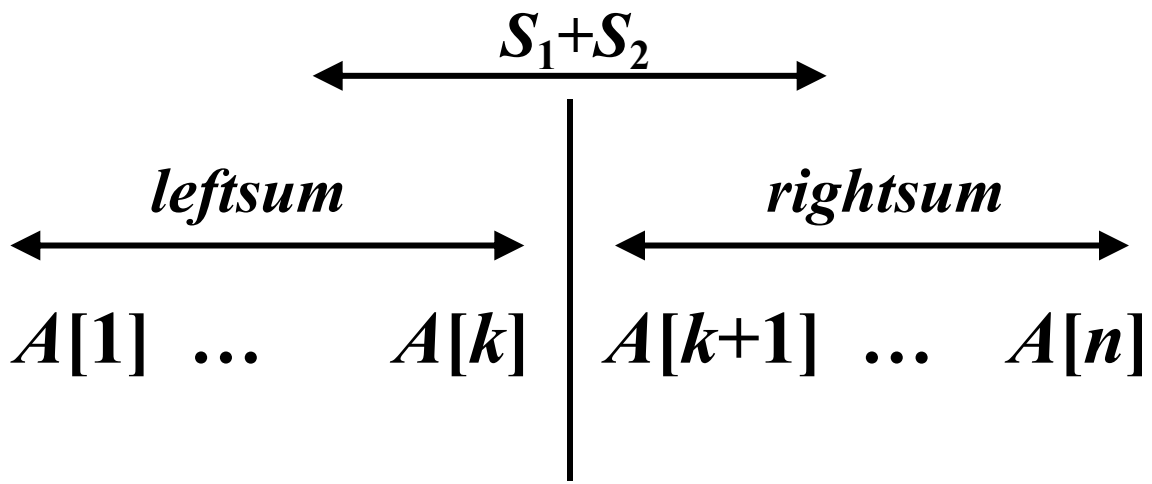
将序列分成左右两半，中间分点*center*

递归计算左段最大子段和 *leftsum*

递归计算右段最大子段和 *rightsum*

$a_{center}\rightarrow a_1$的最大和$S_1$，$a_{center+1}\rightarrow a_n$的最大和$S_2$

max { *leftsum*, *rightsum*, S_1+S_2}



最大子段和 算法2：分治算法

算法 MaxSubSum(*A*, *left*, *right*）

输入：数组*A*，*left*，*right*分别是*A*的左、右边界
输出：*A*的最大子段和*sum*及其子段的前后边界

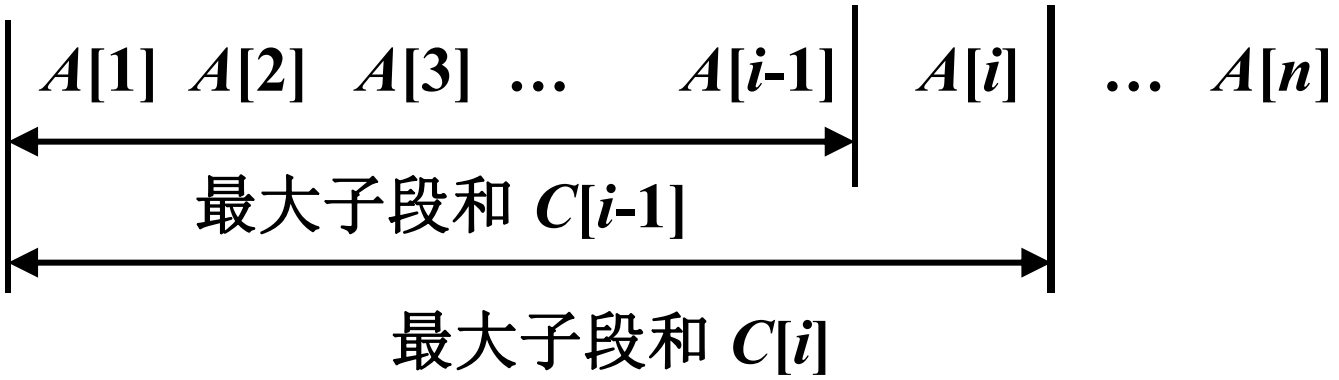
```
1. if |A| = 1 then 输出元素值（当值为负时输出0）
2. center ← ⌊(left+right)/2⌋
3. leftsum ← MaxSubSum(A,left,center)      //子问题A1
4. righsum ← MaxSubSum(A,center+1,right)   //子问题A2
5. S1 ← A1[center]向左的最大和  //从center向左的最大和
6. S2 ← A2[center+1]向右的最大和  //从center+1向右的最大和
7. sum ← S1+S2
8. if  leftsum > sum  then  sum ← leftsum
9. if  rightsum >sum  then  sum ← rightsum
```

时间：$T(n)=2T(n/2)+O(n)$, $T(c)=O(1)$
$T(n)=O(n\log n)$

最大子段和 算法3：动态规划

令$C[i]$ 是 $A[1..i]$中必须包含元素$A[i]$的最大子段和

$$C[i] = \max_{1 \le k \le i} \{\sum_{j=k}^{i} A[j]\}$$



递推方程: $C[i]= \max\{C[i-1]+A[i], A[i]\}$ $i=2,\ldots,n$

$C[1]=A[1]$

解: $\mathrm{OPT}(A) = \max\{0, \max_{0 \le i \le n}\{C[i]\}\}$

最大子段和 算法3：动态规划

算法 MaxSum(*A*, *n*)

输入：数组*A*

输出：最大子段和*sum*, 子段的最后位置*c*

```
1.  sum ← 0
2.  b ← 0   // b是前一个最大子段和
3.  for  i ← 1  to  n  do
4.      if  b>0
5.      then  b ← b+A[i]
6.      else  b ← A[i]
7.      if  b>sum
8.      then  sum ← b
9.            c ← i   // 记录最大和的末项标号
10. return sum, c
```

正确性：对 n 做数学归纳法证明

时间复杂度*O*(*n*)；空间复杂度*O*(*n*)

子集和（subset sum）

- 给定任务 {1, 2, …, n}，各个任务的执行时间分别是 w_i ($1 \le i \le n$)。任务可以在 [0,W] 时间内在一台机器调度执行。求使该机器利用率尽可能高的调度方案。

$$max_{S \subseteq \{1,2,\cdots,n\}} \sum_{i \in S} w_i$$

$$s.t. \quad \sum_{i \in S} w_i \le W$$

（背包问题的特例）

背包问题

- 给定 n 个物品和一个“背包”
  - 物品 i 重量 $w_i > 0$ 千克、价值 $v_i > 0$ 元
  - 背包容量是 W 千克
- 求：背包能容纳的总价值最大的物品列表
  - 如右侧例子：OPT 是物品 { 3, 4 }，总价值 40 元
  - 物品不能切割：“0-1 背包”

W = 11

物品	价值	重量
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

- 贪心：持续挑选能放进背包的单位重量价值 v_i/w_i 最高的物品
  - 例如：{ 5, 2, 1 } 总价值 35 元，非最优方案

背包问题：尝试动态规划

- 令 $OPT(i)$ = 前 i 项物品 $1, \ldots, i$ 总重量 $\leq W$ 子集的最大价值

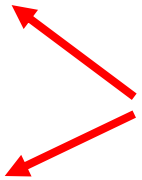
- Case 1：$OPT(i)$ 放弃物品 i
  - $OPT(i) = OPT(i-1)$
- Case 2：$OPT(i)$ 保留物品 i
  - 保留物品 i 不能推断是否需要放弃其他物品
  - 问题无法归约到 $OPT(i-1)$，因为此时需要挑选总重量 $\leq W - w_i$ 的最大价值子集

- 结论：需要构造更多子问题，以增强归纳法论证中的归纳假设

背包问题：动态规划（增强归纳假设）

- 令 $OPT(i, w)$ = 前 i 项物品 $1, \ldots, i$ 总重量 $\leq w$ 子集的最大价值

- Case 1： $OPT(i, w)$ 保留物品 i
  - $OPT(i, w) = v_i + OPT(i-1, w-w_i)$



选择更高价值的方案

- Case 2： $OPT(i, w)$ 放弃物品 i
  - $OPT(i, w) = OPT(i-1, w)$

- 综上，

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

背包问题：动态规划伪代码

记忆化递归版

```
Comp-OPT(i,w)
    if M[i,w] == empty
      if (i==0)
        M[i,w]=0
      else if (w_i > w)
        M[i,w]= Comp-OPT(i-1,w)
      else
        M[i,w]= max {Comp-OPT(i-1,w), v_i + Comp-OPT(i-1,w-w_i)}
    return M[i, w]
```

非递归版

```
for w = 0 to W
    M[0, w] = 0
for i = 1 to n
    for w = 1 to W
       if (w_i > w)
          M[i, w] = M[i-1, w]
       else
          M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}

return M[n, W]
```

背包问题：动态规划

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0											
{ 1, 2 }	0											
{ 1, 2, 3 }	0											
{ 1, 2, 3, 4 }	0											
{ 1, 2, 3, 4, 5 }	0											

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

```
if (w_i > w)
  M[i, w] = M[i-1, w]
else
  M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}
```

背包问题：动态规划

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0											
{ 1, 2, 3 }	0											
{ 1, 2, 3, 4 }	0											
{ 1, 2, 3, 4, 5 }	0											

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

```
if (w_i > w)
  M[i, w] = M[i-1, w]
else
  M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}
```

背包问题：动态规划

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7								
{ 1, 2, 3 }	0	1										
{ 1, 2, 3, 4 }	0	1										
{ 1, 2, 3, 4, 5 }	0	1										

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

```
if (wi > w)
  M[i, w] = M[i-1, w]
else
  M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}
```

背包问题：动态规划

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19					
{ 1, 2, 3, 4 }	0	1										
{ 1, 2, 3, 4, 5 }	0	1										

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

```
if (wi > w)
  M[i, w] = M[i-1, w]
else
  M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}
```

背包问题：动态规划

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29		
{ 1, 2, 3, 4, 5 }	0	1										

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

```
if (wi > w)
  M[i, w] = M[i-1, w]
else
  M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}
```

背包问题：动态规划

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

```
if (w_i > w)
  M[i, w] = M[i-1, w]
else
  M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}
```

背包问题：运行时间

- 运行时间 $O(n \cdot W)$
  - 并非输入规模的多项式函数!
  - ”伪多项式”

- 背包问题的判定问题版本是 NP 完全的（第11讲）

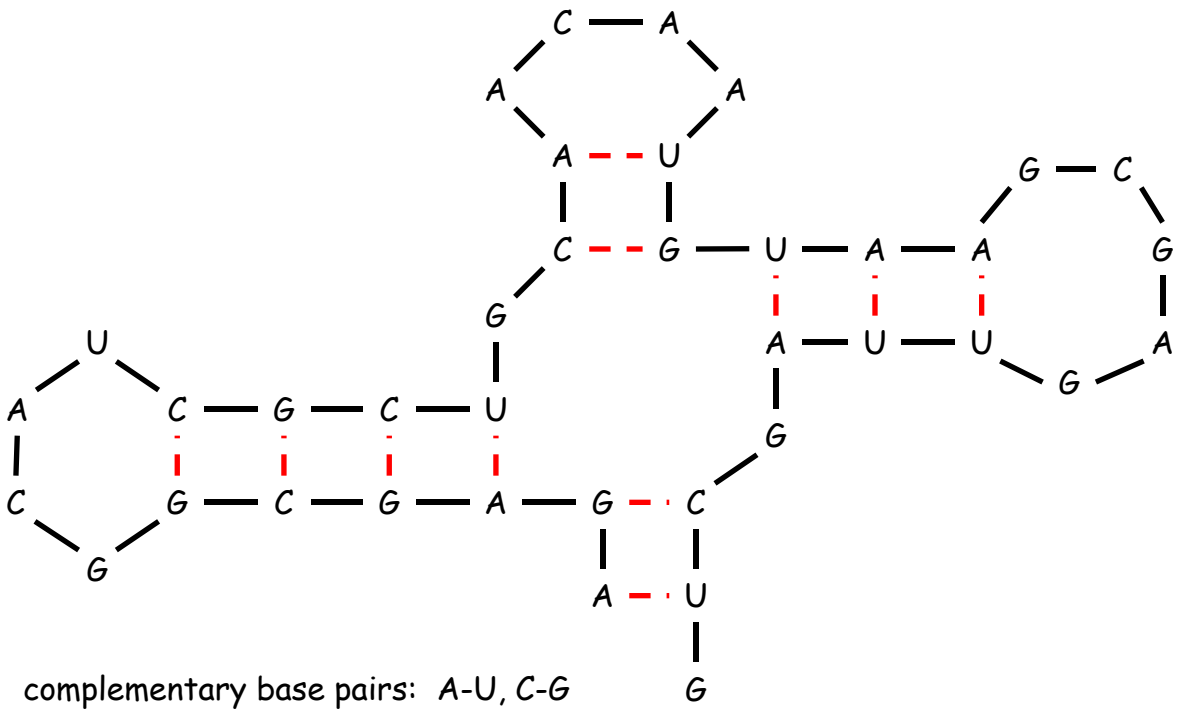
- 背包问题近似算法（第12讲）
  - 存在多项式时间算法，求出与最优值相差不超过 0.01% 的可行解

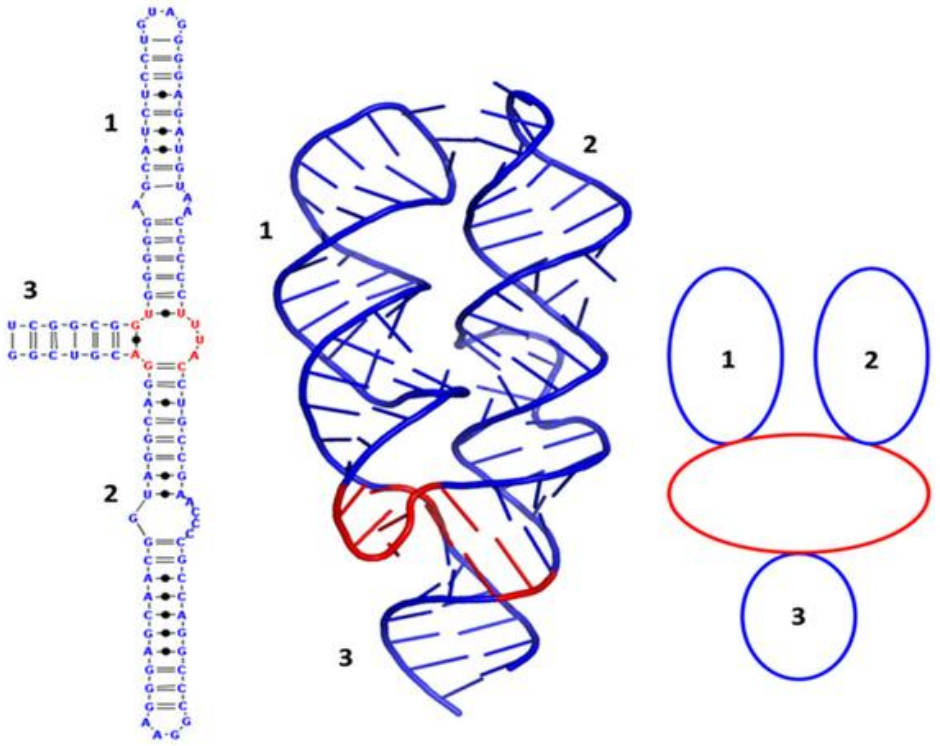
RNA 二级结构

RNA：字母表 { A, C, G, U } 的字符串 $B = b_1 b_2 \ldots b_n$

二级结构：RNA 单链与自身碱基互补配对，形成自我折叠。分析这种结构对于理解 RNA 分子的行为至关重要。

Ex: **GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA**





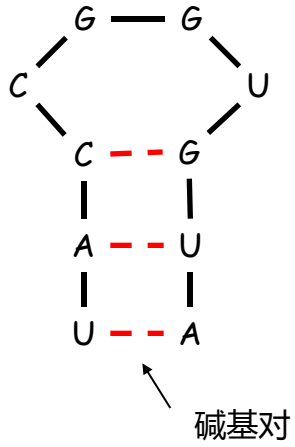
RNA 二级结构：问题形式化

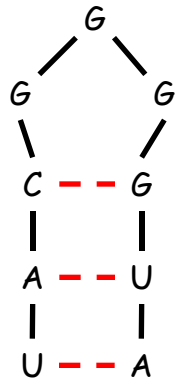
- 二级结构：满足以下条件的碱基配对集合 $S = \{(b_i, b_j)\}$
  - [匹配]：S 是一个“匹配”（每个碱基最多只出现在一组配对中）
  - [有效]：S 的碱基配对只有 (A,U), (U,A), (C,G), (G,C) 四种形式
  - [“钝角”]：每对匹配的碱基至少相隔4个碱基，即 If $(b_i, b_j) \in S$, then $i < j - 4$
  - [不交叉]：如果 (b_i, b_j) 和 (b_k, b_l) 是 S 两对碱基，不会出现 $i < k < j < l$ 的情况
- 自由能：通常假设 RNA 分子最大化总自由能

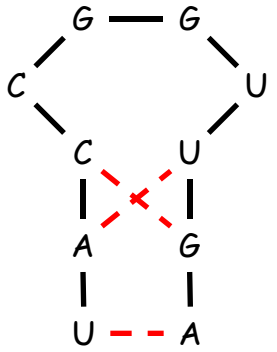
近似为碱基配对的数目

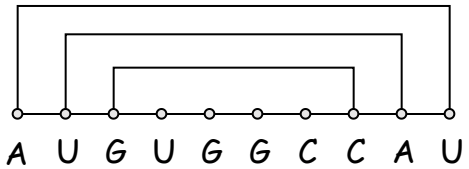
- 目标：给定 RNA 分子 B = $b_1 b_2 ... b_n$，求最大化碱基配对的二级结构

RNA 二级结构：例子

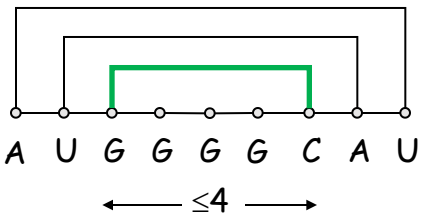




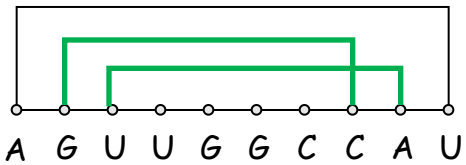




合法



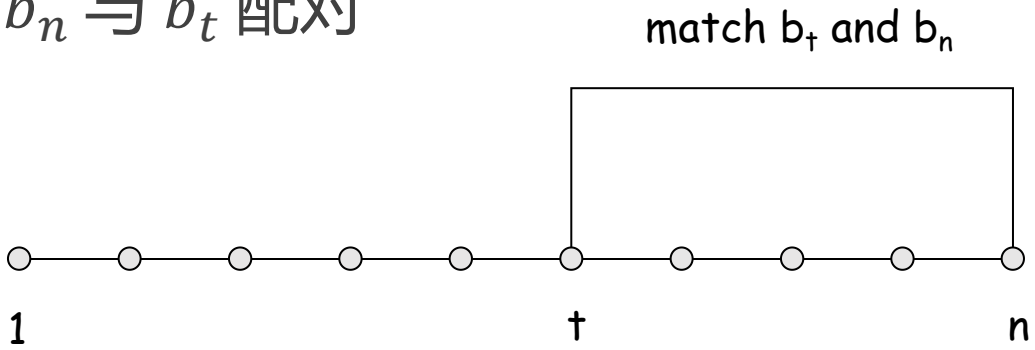
锐角



交叉

RNA 二级结构：尝试动态规划

- 尝试：令 $OPT(n)$ = RNA $b_1b_2...b_n$ 二级结构所包含碱基配对的最大数目
- 假设在 $OPT(n)$ 中，b_n 与 b_t 配对



- 困难：分解为两个子问题
  - 求 $b_1, ..., b_{t-1}$ 的二级结构，即 OPT(t-1)
  - 求 $b_{t+1}, ..., b_{n-1}$ 的二级结构，🤔？🤔？🤔？

RNA 二级结构：再次尝试动态规划

- 令 $OPT(i,j)$ = RNA 片段 $b_i, b_{i+1}, \ldots, b_j$ 二级结构所包含碱基的最大配对数

正确动规的要点；解决了上页的疑惑

- Case 1： $j - i \leq 4$
  - 根据“钝角”条件，$OPT(i,j) = 0$
- Case 2：碱基 b_j 不属于任一配对
  - $OPT(i,j) = OPT(i,j-1)$
- Case 3：碱基 b_j 与 b_t 配对，其中 $i \leq t < j-4$
  - 利用“不交叉”条件将子问题解耦合
  - $OPT(i,j) = \max_{i \leq t < j-4} \{ 1 + OPT(i, t-1) + OPT(t+1, j-1) \}$

RNA 二级结构：记忆化递归代码

```
Let M[i,j]=empty for all i,j.

Compute-OPT(i,j){
    if (j-i <= 4)
      return 0;
    if (M[i,j] is empty)
       M[i,j]=Compute-OPT(i,j-1)
       for t=i to j-5 do
          if ((b_t, b_j) is in {(A,U),(U,A),(C,G),(G,C)})
            M[i,j]=max(M[i,j], 1+Compute-OPT(i,t-1) +
             Compute-OPT(t+1,j-1))
    return M[i,j]
}
```

算法能终止吗?

RNA 二级结构：归纳证明

令 $OPT(i,j)$ = RNA 片段 $b_i, b_{i+1}, \ldots, b_j$ 二级结构所包含碱基的最大配对数

归纳基础：对任意 $|j-i| \le 4$，有 $OPT(i,j) = 0$

归纳假设：对某 $\ell \ge 4$，假设已正确计算出所有 $|i-j| \le \ell$ 的 $OPT(i,j)$

归纳推理：待证明对任意 $|i-j| = \ell + 1$，能正确计算 $OPT(i,j)$。

以下 是某对 i,j 满足 $|i-j| = \ell + 1$ 的情况

Case 1：碱基 b_j 不属于任一配对

- $OPT(i,j) = OPT(i,j-1)$ [根据归纳假设，右值参数满足 $|i-(j-1)| = \ell$，已有正确值]

Case 2：碱基 b_j 与 b_t 配对，其中 $i \le t < j-4$

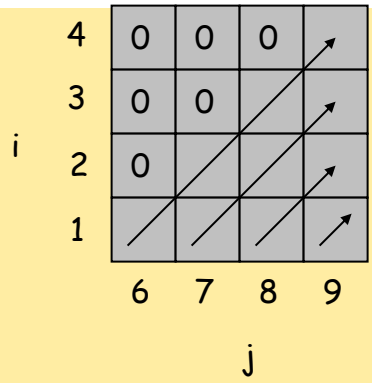
- $OPT(i,j) = \max_{i \le t < j-4} \{ 1 + OPT(i, t-1) + OPT(t+1, j-1) \}$

两组参数均满足归纳假设的条件（$\le \ell$）

RNA 二级结构：自底向上动规

```
for ℓ = 1, 2, …, n-1
  for i = 1, 2, …, n-1
    j = i + ℓ
    if (ℓ <= 4)
      M[i,j]=0;
      else
        M[i,j]=M[i,j-1]
        for t=i to j-5 do
          if (b_t, b_j is in {A-U, U-A, C-G, G-C})
            M[i,j]=max(M[i,j], 1+ M[i,t-1] + M[t+1,j-1])

  return M[1, n]
}
```



运行时间：$O(n^3)$
(沿 i, j 或 j, i 循环皆可)

序列比对 / 编辑距离

- 编辑距离：给定两个序列 S1 和 S2，通过一系列字符编辑（插入、删除、替换）操作，将S1转变成S2. 完成这种转换所需要的最少的编辑操作个数称为 S1 和 S2 的编辑距离.
- 实例：vintner 转变成 writers，编辑距离 6：

```
          vintner
删除v：   -intner
插入w：   wintner
插入r：   wrintner
删除n：   wri-tner
删除n：   writ-er
插入s：   writers
```

序列比对 / 编辑距离

- 编辑距离应用
  - 核苷酸的序列比对
  - diff
  - git
  - 拼写检查
  - 抄袭检查
  - ...

- 序列比对应用：生物信息学

  核苷酸的序列比对

```
AGGCTATCACCTGACCTCCAGGCCGATGCCC
TAGCTATCACGACCGCGGTCGATTTGCCCGAC
```

  比对结果

```
-AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---
TAG-CTATCAC--GACCGC--GGTCGATTTGCCCGAC
```

序列比对：算法设计

- S1[1..n] 和 S2[1..m] 表示两个子序列
- 子问题划分：S1[1..i] 和 S2[1..j]
- C[i,j]： S1[1..i] 和 S2[1..j] 的编辑距离

$$C[i,j]=\min\{C[i-1,j]+1, C[i,j-1]+1, C[i-1,j-1]+t[i,j]\}$$

$$t[i,j]=\begin{cases}0 & S_1[i]=S_2[j]\\ 1 & S_1[i]\neq S_2[j]\end{cases}$$

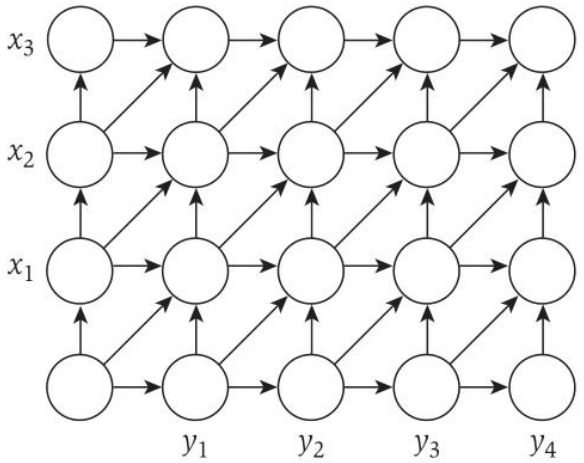
$$C[0,j]=j,$$

$$C[i,0]=i$$

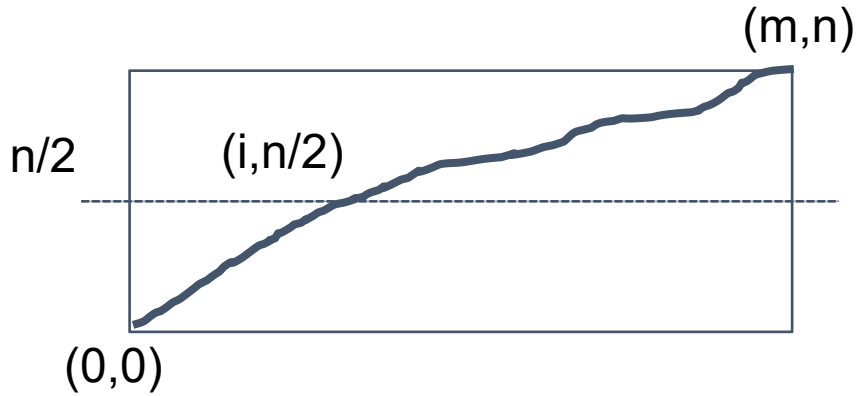
- 算法的时间复杂度是O(nm)

序列比对：算法设计

- O(mn) 空间?
  - 如果不需要重建解，
    O(min(m,n)) 空间即可

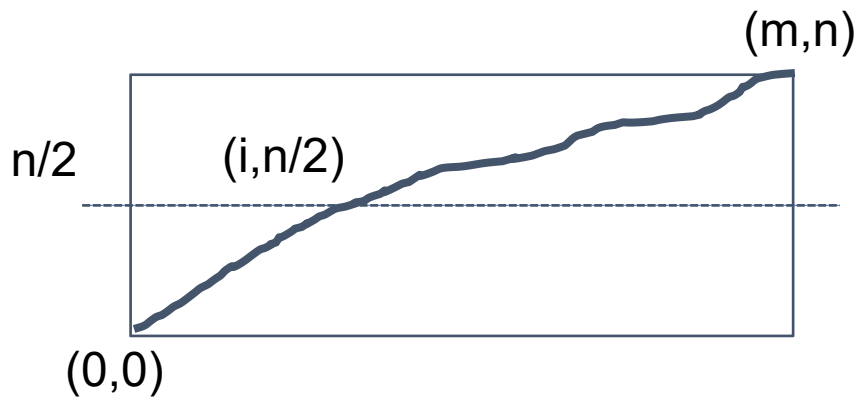


- 可用分治，降至 O(m+n)

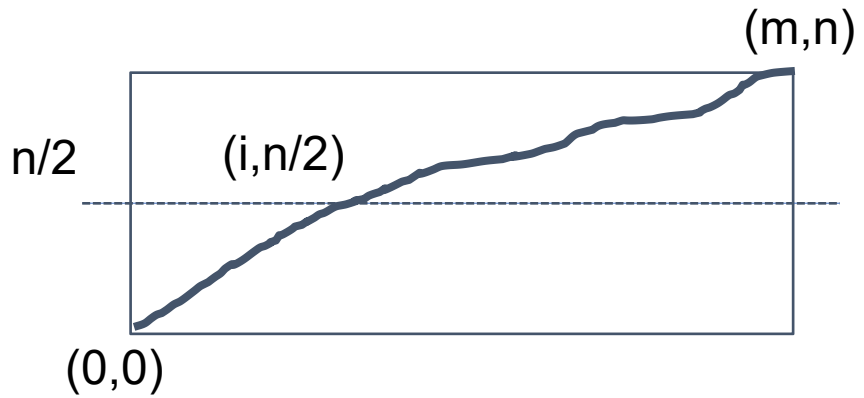


线性空间的序列比对算法

- Hirschberg算法
  - D. S. Hirschberg. 1975. A linear space algorithm for computing maximal common subsequences. Commun. ACM 18, 6 (June 1975), 341–343.
  - 结合分治策略和动态规划算法
- 序列比对问题的性质
  - 性质1：假如从(0,0)到(m,n)的最短路径经过(i,j)，则长度为 f(i,j)+g(i,j)
    - 其中 f(i,j) 为 (0,0) 到 (i,j) 的最短路径长度，g(i,j) 为 (i,j) 到 (m,n) 的最短路径长度
  - 性质2：令 q 为最小化 f(q,n/2)+g(q,n/2) 的下标，则 (0,0) 到 (m,n) 存在经过 (q,n/2) 的最短路径



线性空间的序列比对算法



- 空间复杂度分析
  - 需要 Θ(m) 的空间维护 f(*,n/2) 和 g(*,n/2)
  - 递归次数 ≤ n，每次递归调用需要 Θ(1) 空间
  - 空间复杂度 Θ(m+n)
- 时间复杂度分析
  - 求解以下递推方程，可用数据归纳法证明 T(m,n) = Θ(mn)
  - T(i,j) = cmn + T(q, n/2) + T(m-q, n/2)
  - T(m,2) ≤ cm
  - T(2,n) ≤ cn

动态规划算法设计步骤

- 将问题表示成多步判断，确定子问题的边界
  - 整个判断序列就对应问题的最优解
  - 每步判断对应一个子问题，子问题类型与原问题一样
- 确定优化函数，以函数的极大（或极小）作为判断的依据
- 确定是否满足优化原则——动态规划算法的必要条件
- 确定子问题的重叠性
- 列出关于优化函数的递推方程（或不等式）和边界条件
  - 正确性的证明
- 自底向上计算子问题的优化函数值
- 备忘录方法（表格）存储中间结果
- 设立标记函数 s[i,j] 求解问题的解