动态规划算法

从斐波那契数列看动态规划

- 斐波那契数列：$F_n = F_{n-1} + F_{n-2}$
- 练习：使用**递归**和**非递归**的方法来求解斐波那契数列的第n项

```
# 子问题的重复计算
def fibnacci(n):
    if n == 1 or n == 2:
        return 1
    else:
        return fibnacci(n-1) + fibnacci(n-2)
```

```
fibnacci(40)
```

```
102334155
```

```
# 动态规划（DP）的思想 = 递推式
def fibnacci_no_recursion(n):
    f = [0,1,1]
    if n > 2:
        for i in range(n-2):
            num = f[-1] + f[-2]
            f.append(num)
    return f[n]
```

```
fibnacci_no_recursion(40)
```

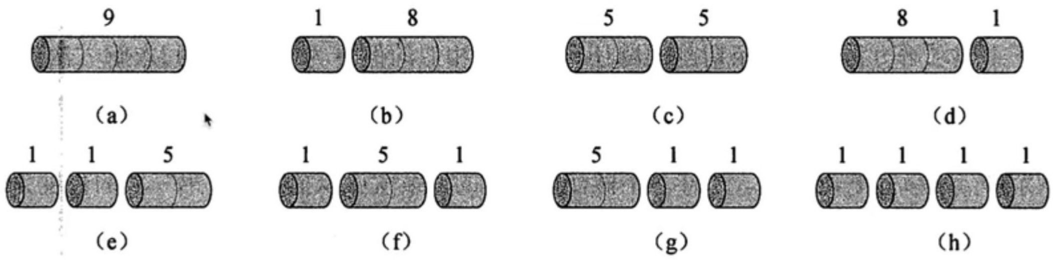
```
102334155
```

钢条切割问题

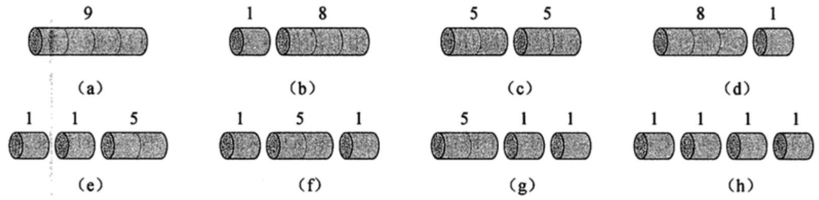
- 某公司出售钢条，出售价格与钢条长度之间的关系如下表：

长度i	1	2	3	4	5	6	7	8	9	10
价格p_i	1	5	8	9	10	17	17	20	24	30

- 问题：现有一段长度为n的钢条和上面的价格表，求切割钢条的方案，使得总收益最大。



- 思考：长度为n的钢条的不同切割方案有几种？
    - 有2^{n-1}次，因为长度为n的钢条有n-1个切割口，每个切割口有两种选择：切或不切



长度i	1	2	3	4	5	6	7	8	9	10
价格p_i	1	5	8	9	10	17	17	20	24	30

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30

递推式

- 设长度为n的钢条切割后最优收益值为r_n,可以得出递推式：
    - $r_n = max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$
- 第一个参数p_n表示不切割
- 其他n-1个参数分别表示另外n-1种不同切割方案，对方案 i=1,2,...,n-1
    - 将钢条切割为长度为i和n-i两段
    - 方案i的收益为切割两段的最优收益之和
- 考察所有的i，选择其中收益最大的方案

最优子结构

- 可以将求解规模为n的原问题，划分为规模更小的子问题：完成一次切割后，可以将产生的两段钢条看成两个独立的钢条切割问题。
- 组合两个子问题的最优解，并在所有可能的两段切割方案中选取组合收益最大的，构成原问题的最优解。
- 钢条切割满足**最优子结构**：问题的最优解由相关子问题的最优解组合而成，这些子问题可以独立求解。
- 钢条切割问题还存在更简单的递归求解方法:
    - 从钢条的左边切割下长度为i的一段，只对右边剩下的一段继续进行切割，左边的不再切割
    - 递推式**简化**为 $r_n = \max\limits_{1 \le i \le n}(p_i + r_{n-i})$
    - 不做切割的方案就可以描述为：左边一段长度为n，收益为p_n，剩余一段长度为0，收益为$r_0 = 0$

自顶向下递归实现

```
p = [0,1,5,8,9,10,17,17,20,21,23,24,26,27,27,28,30,33,36,39,40]
#p = [0,1,5,8,9,10,17,17,20,24,30]
```

```
def cut_rod_recursion(p, n):
    if n == 0:
        return 0
    else:
        res = p[n]
        for i in range(1, n):
            res = max(res, cut_rod_recursion(p, i) + cut_rod_recursion(p, n-i))
        return res
```

上面代码缺点：重复计算+递归

```
cut_rod_recursion(p, 12)
```

```
34
```

```
def cut_rod_recursion_2(p, n):
    if n == 0:
        return 0
    else:
        res = 0
        for i in range(1, n+1):
            res = max(res, p[i]+cut_rod_recursion_2(p, n-i))
        return res
```

```
cut_rod_recursion_2(p, 12)
```

```
34
```

- 为何自顶向下递归实现的效率会这么差?
    - 时间复杂度 $O(2^n)$

动态规划解法

- 递归算法由于重复求解相同子问题，效率极低
- 动态规划的思想：
    - 每个子问题只求解一次，保存求解结果
    - 之后需要此问题时，只需查找保存的结果

```python
def cut_rod_dp(p, n):
    r = [0]
    for i in range(1, n+1): # 公式里面的n
        res = 0
        for j in range(1, i+1): # 公式里面的i
            res = max(res, p[j]+r[i-j])
        r.append(res)
    return r[n]
```

```python
cut_rod_dp(p, 20)
```

```
56
```

```python
def cut_rod_dp(p, n):
    r = [0 for _ in range(n+1)]
    for j in range(1, n+1):
        q = 0
        for i in range(1, j+1):
            q = max(q, p[i]+r[j-i])
        r[j] = q
    return r[n]
```

重构解

- 如何修改动态规划算法，使其不仅输出最优解，还输出最优切割方案？
- 对每个子问题，保存切割一次时左边切下的长度

```python
def cut_rod_extend(p, n):
    r = [0]
    s = [0]
    for i in range(1, n+1):
        res_r = 0 # 记录价格最大值
        res_s = 0 # 价格最大值对应方案左边不切割长度
        for j in range(1, i+1): # 公式里面的i
            if p[j] + r[i-j] > res_r:
                res_r = p[j] + r[i-j]
                res_s = j
        r.append(res_r)
        s.append(res_s)
    return r[n], s
```

```python
r,s = cut_rod_extend(p,10)
```

```python
s
```

```
[0, 1, 2, 3, 2, 2, 6, 1, 2, 3, 10]
```

```python
def cut_rod_solution(p, n):
    r, s = cut_rod_extend(p, n)
    ans = []
    while n > 0:
        ans.append(s[n])
        n -= s[n]
    return ans
```

```python
cut_rod_solution(p,20)
```

```
[2, 6, 6, 6]
```

```python
cut_rod_dp(p,20)
```

```
56
```

```python
r,s = cut_rod_extend(p,20)
```

```python
s
```

```
[0, 1, 2, 3, 2, 2, 6, 1, 2, 3, 2, 2, 6, 1, 2, 3, 2, 2, 6, 1, 2]
```

动态规划问题关键特征

- 什么问题可以使用动态规划方法?
  - 最优子结构
    - 原问题的最优解中涉及多少个子问题
    - 再确定最优解使用哪些子问题时，需要考虑多少种选择
  - 重叠子问题

最长公共子序列

- 一个序列的子序列是在该序列中删去若干元素后得到的序列。
  - 例：“ABCD”和“BDF”都是“ABCDEFG”的子序列
- 最长公共子序列（LCS）问题：给定两个序列X和Y，求X和Y长度最大的公共子序列。
  - 例：X='ABBCBDE',Y='DBBCDB',LCS(X,Y)='BBCD'
- 应用场景：字符串相似度比对
- 思考：暴力穷举法的时间复杂度--> $O(2^n)$
- 思考：最长公共子序列是否具有最优子结构性质?
- 例如：要求a="ABCBDAB"与b="BDCABA"的LCS:
  - 由于最后一位“B”~=“A”

- 因此LCS(a,b)应该来源于LCS(a[:-1],b)与LCS(a,b[:-1])中更大的那一个

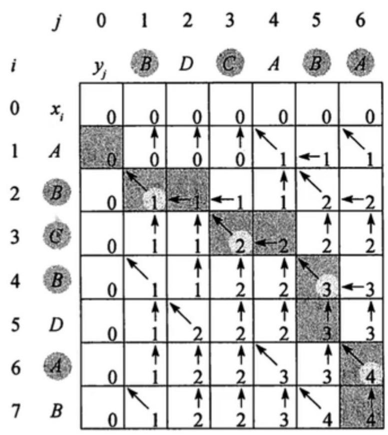
定理 15.1（LCS 的最优子结构） 令 $X=\langle x_1, x_2, \cdots, x_m\rangle$ 和 $Y=\langle y_1, y_2, \cdots, y_n\rangle$ 为两个序列，$Z=\langle z_1, z_2, \cdots, z_k\rangle$ 为 X 和 Y 的任意 LCS。

1. 如果 $x_m=y_n$，则 $z_k=x_m=y_n$ 且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的一个 LCS。
2. 如果 $x_m\neq y_n$，那么 $z_k\neq x_m$ 意味着 Z 是 X_{m-1} 和 Y 的一个 LCS。
3. 如果 $x_m\neq y_n$，那么 $z_k\neq y_n$ 意味着 Z 是 X 和 Y_{n-1} 的一个 LCS。

- 最优解的递推式：

$$c[i,j]=\begin{cases}0 & 若\ i=0\ 或\ j=0\\ c[i-1,j-1]+1 & 若\ i,j>0\ 且\ x_i=y_j\\ \max(c[i,j-1],c[i-1,j]) & 若\ i,j>0\ 且\ x_i\neq y_j\end{cases}$$

  - c[i,j]表示X_i和Y_j的LCS长度



$$c[i,j]=\begin{cases}0 & 若\ i=0\ 或\ j=0\\ c[i-1,j-1]+1 & 若\ i,j>0\ 且\ x_i=y_j\\ \max(c[i,j-1],c[i-1,j]) & 若\ i,j>0\ 且\ x_i\neq y_j\end{cases}$$

```python
def lcs_length(x, y):
    m = len(x)
    n = len(y)
    c = [[0 for _ in range(n+1)] for _ in range(m+1)]
    for i in range(1, m+1):
        for j in range(1, n+1):
            if x[i-1] == y[j-1]: # i j 位置上的字符匹配时，来自左上方+1
                c[i][j] = c[i-1][j-1] + 1
            else:
                c[i][j] = max(c[i-1][j],c[i][j-1])
    for _ in c:
        print(_)
    return c[m][n]
```

```python
lcs_length("ABCBDAB",'BDCABA')
```

```
[0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 1, 1]
[0, 1, 1, 1, 1, 2, 2]
[0, 1, 1, 2, 2, 2, 2]
[0, 1, 1, 2, 2, 3, 3]
[0, 1, 2, 2, 2, 3, 3]
[0, 1, 2, 2, 3, 3, 4]
[0, 1, 2, 2, 3, 4, 4]
```

```
4
```

```python
def lcs(x, y):
    m = len(x)
    n = len(y)
    c = [[0 for _ in range(n+1)] for _ in range(m+1)]
    b = [[0 for _ in range(n+1)] for _ in range(m+1)] # 1 左上方 2 上方 3 左方
    for i in range(1, m+1):
        for j in range(1, n+1):
            if x[i-1] == y[j-1]: # i j 位置上的字符匹配时，来自左上方+1
                c[i][j] = c[i-1][j-1] + 1
                b[i][j] = 1
            elif c[i-1][j] > c[i][j-1]:
                c[i][j] = c[i-1][j]
                b[i][j] = 2
            else:
                c[i][j] = c[i][j-1]
                b[i][j] = 3
    return c[m][n], b
```

```python
c,b = lcs("ABCBDAB",'BDCABA')
for _ in b:
    print(_)
```

```
[0, 0, 0, 0, 0, 0, 0]
[0, 3, 3, 3, 1, 3, 1]
[0, 1, 3, 3, 3, 1, 3]
[0, 2, 3, 1, 3, 3, 3]
[0, 1, 3, 2, 3, 1, 3]
[0, 2, 1, 3, 3, 2, 3]
[0, 2, 2, 3, 1, 3, 1]
[0, 1, 2, 3, 2, 1, 3]
```

```python
def lcs_trackback(x, y):
    c, b = lcs(x,y)
    i = len(x)
    j = len(y)
    res = []
    while i > 0 and j > 0:
        if b[i][j] == 1:  # 来自左上方->匹配
            res.append(x[i-1])
            i -= 1
            j -= 1
        elif b[i][j] == 2: # 来自上方->不匹配
            i -= 1
        else: # 来自左方
            j -= 1
    return ''.join(reversed(res))
```

```python
lcs_trackback("ABCBDAB",'BDCABA')
```

```
'BDAB'
```

欧几里得算法

最大公约数

- 约数：如果整数a能被整数b整除，那么a叫做b的倍数，b叫做a的约数。
- 给定两个整数a,b，两个数的所有公约数中最大值即为最大公约数（Greatest Common Divisor, GCD）
- 例：12和16的最大公约数是4
- 如何计算两个数的最大公约数：
    - 欧几里得：**辗转相除法（欧几里得算法）**
    - 《九章算术》：更相减损术
- 欧几里得算法：gcd(a,b) = gcd(b, a mod b)
    - 例：gcd(60,21) = gcd(21,18) = gcd(18,3) = gcd(3,0) = 0

```python
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)
```

```python
gcd(12,16)
```

```
4
```

```python
def gcd2(a, b):
    while b > 0:
        r = a % b
        a = b
        b = r
    return a
```

```python
gcd2(12,16)
```

```
4
```

实现分数计算

```
class Fraction:
    def __init__(self, a, b):
        self.a = a
        self.b = b
        x = self.gcd(a, b)
        self.a /= x
        self.b /= x

    def gcd(self, a, b):
        while b > 0:
            r = a % b
            a = b
            b = r
        return a

    def zgs(self, a, b): # 最小公倍数
        # 12 16 --> 4
        # 3*4*4 = 48
        x = self.gcd(a, b)
        return a * b / x #x *(a / x) * (b / x)

    def __add__(self, other):
        # 1/12 + 1/20
        a = self.a
        b = self.b
        c = other.a
        d = other.b
        denom = self.zgs(b, d)
        mem = a * (denom / b) + c * (denom / d)
        return Fraction(mem, denom)

    def __repr__(self):
        return '%d/%d' % (self.a, self.b)
```

```
f = Fraction(30,10)
f
```

```
3/1
```

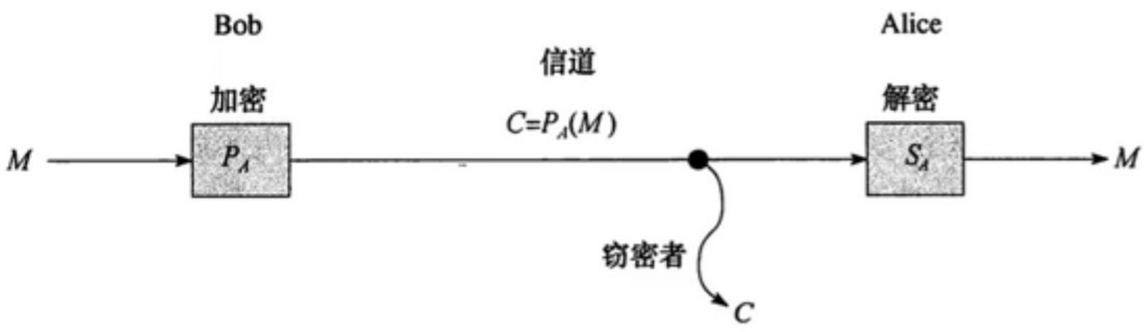
```
a = Fraction(1,3)
b = Fraction(1,2)
a+b
```

```
5/6
```

RSA算法

密码与加密

- 传统密码：加密算法是秘密的
- 现代密码传统：加密算法是公开的，密钥是秘密的
    - 对称加密：加密解密同一个密钥
    - 非对称加密：加密解密不同密钥
- RSA非对称加密系统：
    - 公钥：用来加密，是公开的
    - 私钥：用来解密，是私有的



RSA加密算法过程

1. 随机选取两个质数p和q
2. 计算n = pq（n很难分解成p和q，因此很难破解，如果破解只是破解了一对密钥，更换后仍需重新破解）
3. 选取一个与$\phi(n)$互质的小奇数e，$\phi(n) = (p-1)(q-1)$
4. 对模$\phi(n)$，计算e的乘法逆元d，即满足$(e*d) mod \phi(n) = 1$
5. 公钥(e,n) 私钥(d,n)

- 加密过程：$c = (m^e) mod n$
- 解密过程：$m = (c^d) mod n$