2.1 斐波那契系列问题

在数学上，斐波纳契数列以如下被以递归的方法定义：F(0)=0，F(1)=1, F(n)=F(n-1)+F(n-2)（n>=2，n∈N*）根据定义，前十项为 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

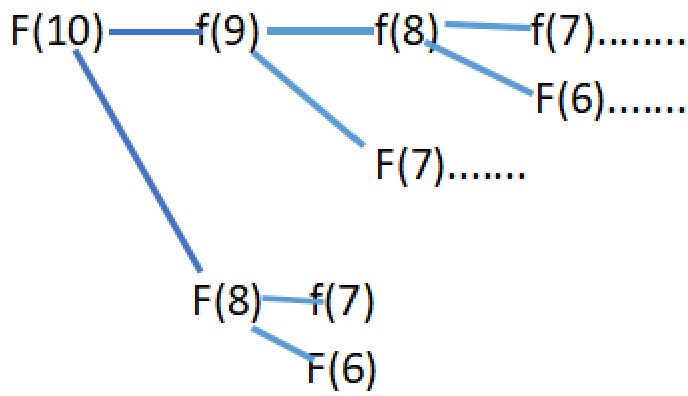
例 1：给定一个正整数 n，求出斐波那契数列第 n 项（这时 n 较小）

解法一：完全抄定义

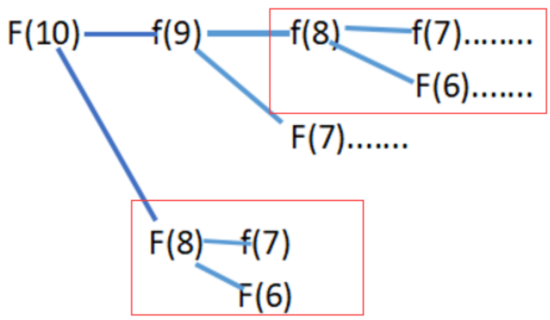
```
def f0(n):
    if n==1 or n==2:
        return 1
    return f(n-1)+f(n-2)
```

分析一下，为什么说递归效率很低呢？咱们来试着运行一下就知道了：

比如想求 f(10)，计算机里怎么运行的？



想算出 f(10)，就要先算出 F(9),
想算出 f(9)，就要先算出 F(8),
想算出 f(8)，就要先算出 F(7),
想算出 f(7)，就要先算出 F(6)……
兜了一圈，我们发现，有相当多的计算都重复了，比如红框部分：



那如何解决这个问题呢？问题的原因就在于，我们算出来某个结果，并没有记录下来，导致了重复计算。那很容易想到如果我们把计算的结果全都保存下来，按照一定的顺序推出 n 项，就可以提升效率

解法 2：

```
def f1(n):
    if n==1 or n==2:
        return 1
    l=[0]*n                    #保存结果
    l[0],l[1]=1,1              #赋初值
    for i in range(2,n):
        l[i]=l[i-1]+l[i-2]     #直接利用之前结果
    return l[-1]
```

可以看出，时间 o(n)，空间 o(n)。
继续思考，既然只求第 n 项，而斐波那契又严格依赖前两项，那我们何必记录那么多值浪费空间呢？只记录前两项就可以了。

解法 3：

```
def f2(n):
    a,b=1,1
```

```
    for i in range(n-1):
        a,b=b,a+b
    return a
```

补充：

1）pat、蓝桥杯等比赛原题：求的 n 很大，F(N)模一个数。应每个结果都对这个数取模，否则：第一，计算量巨大，浪费时间；第二，数据太大，爆内存，

2）对于有多组输入并且所求结果类似的题，可以先求出所有结果存起来，然后直接接受每一个元素，在表中查找相应答案

3）此题有快速幂算法，但是碍于篇幅和同学们水平有限，不再叙述，可以自行学习。

例 2：一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。

依旧是找递推关系：

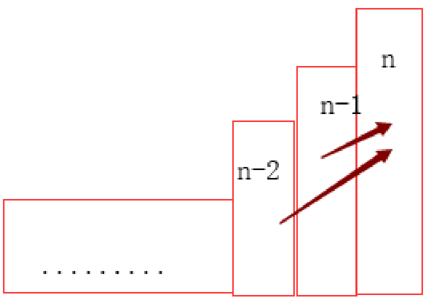
1）跳一阶，就一种方法

2）跳两阶，它可以一次跳两个，也可以一个一个跳，所以有两种

3）三个及三个以上，假设为 n 阶，青蛙可以是跳一阶来到这里，或者跳两阶来到这里，只有这两种方法。

它跳一阶来到这里，说明它上一次跳到 n-1 阶，

同理，它也可以从 n-2 跳过来



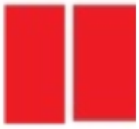
f（n）为跳到 n 的方法数，所以，f(n)=f(n-1)+f(n-2)

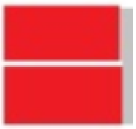
优化思路与例 1 类似，请自行思考。

例 3：我们可以用 2*1 的小矩形横着或者竖着去覆盖更大的矩形。请问用 n 个 2*1 的小矩形无重叠地覆盖一个 2*n 的大矩形，总共有多少种方法？

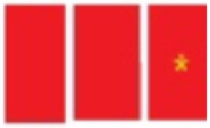
N=1：只有一种

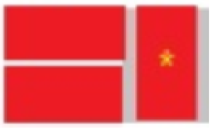
N=2，两种：

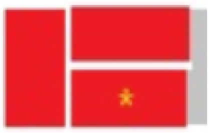
1、


2、


N=3:

1


2


3


读到这里，你们应该能很快想到，依旧是斐波那契式递归啊。

对于 n>=3:怎么能覆盖到三?
只有两种办法，从 n-1 的地方竖着放了一块，或者从 n-2 的位置横着放了两块

例 4：给定一个由 0-9 组成的字符串，1 可以转化成 A，2 可以转化成 B。依此类推。。25 可以转化成 Y，26 可以转化成 z，给一个字符串，返回能转化的字母串的有几种？

比如：123，可以转化成
1 、2 、3 变成 ABC,
12 、3 变成 LC,

1 、23 变成 AW
三种，返回三，

比如 99999，就一种：iiiii，返回一。

分析：求 i 位置及之前字符能转化多少种。

两种转化方法
1）字符 i 自己转换成自己对应的字母
2）和前面那个数组成两位数，然后转换成对应的字母

假设遍历到 i 位置，判断 i-1 位置和 i 位置组成的两位数是否大于 26，大于就没有第二种方法，f(i)=f(i-1),如果小于 26， f(i)=f(i-1)+f(i-2)

2.2 矩阵系列问题

<u>例 5：给一个由数字组成的矩阵，初始在左上角，要求每次只能向下或向右移动，路径和就是经过的数字全部加起来，求可能的最小路径和。</u>

1　3　5　9

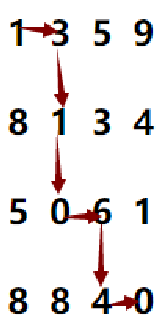
8　1　3　4

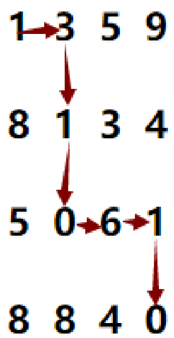
5　0　6　1

8　8　4　0

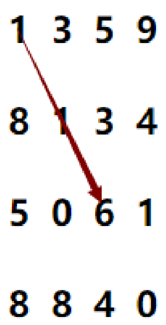
路径：1 3 1 0 6 1 0 路径和最小，返回 12

分析：我们可以像之前一样，暴力的把每一种情况都试一次，但是依旧会造成过多的重复计算，以本题为例子最后解释一下暴力慢在哪里，以后不再叙述了。
比如本题来讲，我们尝试如下路径：

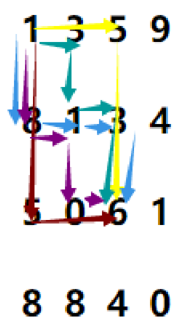




有很多路是重复走过的一遍。
再进一步说：



从 1 到 3 位置，有很多路可以走，直观感受一下：



所有路中，一定会有和最小的，但是我们并不知道，每次尝试一次 1->6->终点的路线时，我们把所有的情况都算了一遍，这过程中我们浪费了相当多的有效信息。

这就是暴力的结果。

优化做法：生成和矩阵相同大小的二维表，用来记录到起点每个位置的最小路径和

	A	B	C	D
1				
2				
3				
4				

接下来带着大家真正进入动态规划；
第一步：初始化（对于本题来说，第一列和第一行，我们别无选择，就一条路，因此，我们可以直接确定答案）

	A	B	C	D
1	1	1+3=4	4+5=9	9+9=18
2	1+8=9			
3	9+5=14			
4	14+8=22			

第二步：确定其余位置如何推出（我们称为状态转移方程）
直观来说，每个位置只可能是从上面，或者左边走来的：

	A	B	C	D
1	1	1+3=4	4+5=9	9+9=18
2	1+8=9			
3	9+5=14			
4	14+8=22			

对于普遍的位置 i，j，只有 i-1,j 和 i,j-1 这两个位置可以一步走到这里，所以 DP[i,j]=min(DP[i,j-1],DP[i-1,j]）+L[i,j]（之前的最优解加上本位置的数字）

继续优化：和之前一样，这个式子实际上也是严格依赖两个值，一个是左边的值，一个是上面的值，所以，我们按之前的思路，应该可以想到可以压缩空间。
我们尝试用一维的空间来解题：

想象这是我们的第一行答案:

1	1	1+3=4	4+5=9	9+9=18

我们如何利用仅有的一维空间来更新出下一行呢?

我们要想:

第一，我们需要左面的数字，所以，本位置的左边必须是更新过的数字（否则就是左上的位置了），所以应该从左往右更新。

第二，我们需要上面的数字，这个不需要更新，本来就需要本位置的旧数字。

本题第二行为：8，1，3，4

第一行答案为

	A	B	C	D
1	1	4	9	18

依次更新:

更新 A:

	A	B	C	D
1	1+8=9	4	9	18

（只能向下走）

更新 B:

	A	B	C	D
1	1+8=9	min(9,4)+1=5	9	18

（比较从左边来和从上面来哪里比较小）

更新 C:

	A	B	C	D
1	1+8=9	5	min(5,9)+3=8	18

更新 D:

	A	B	C	D
1	9	5	8	min(8,18)+4=12

最后我们可以发现，伪代码是这样的:

```
For i   0 ->  高度:
    For j   0 ->  宽度
        DP[j]=min(DP[j-1],DP[j]) +L[i,j]
```

时间不变，空间优化到 o(min（高，宽））

例 6：给一个由数字组成的矩阵，初始在左上角，要求每次只能向下或向右移动，路径和就是经过的数字全部加起来，求可能的最大路径和。

和例 5 只差一个“大”字，请自己思考

例 7：一个矩阵，初始在左上角，要求每次只能向下或向右移动，求到终点的方法数。

和例 5，6 类似，只是方法数应该等于，左边的方法数加上上面的方法数

第二章末练习

1

一个只包含'A'、'B'和'C'的字符串，如果存在某一段长度为 3 的连续子串中恰好'A'、'B'和'C'各有一个，那么这个字符串就是纯净的，否则这个字符串就是暗黑的。例如：

BAACAACCBAAA 连续子串"CBA"中包含了'A','B','C'各一个，所以是纯净的字符串

AABBCCAABB 不存在一个长度为 3 的连续子串包含'A','B','C',所以是暗黑的字符串

你的任务就是计算出长度为 n 的字符串(只包含'A'、'B'和'C')，有多少个是暗黑的字符串。(网易 17 校招原题)

2、X 国的一段古城墙的顶端可以看成 2*N 个格子组成的矩形（如下图所示），现需要把这些格子刷上保护漆。

a	c	e
b	d	f

城墙宽度总是**2**格，长度未知，此时为**3**

你可以从任意一个格子刷起，刷完一格，可以移动到和它相邻的格子（对角相邻也算数），但不能移动到较远的格子（因为油漆未干不能踩！）

比如：a d b c e f 就是合格的刷漆顺序。

c e f d a b 是另一种合适的方案。

当已知 N 时，求总的方案数。当 N 较大时，结果会迅速增大，请把结果对 1000000007 (十亿零七) 取模。

3.1 01 背包

入门了动态规划之后，我们来看一个经典系列问题：背包问题

有N件物品和一个容量为V的背包。第i件物品的费用是$w[i]$，价值是$v[i]$，求将哪些物品装入背包可使价值总和最大。

这是最基础的背包问题，特点是：每种物品仅有一件，可以选择放或不放。

用子问题定义状态：

f[i][j]表示前 i 件物品恰放入一个容量为 j 的背包可以获得的最大价值。则其状态转移方程为：

$$f[i][j] = max(f[i-1][j], f[i-1][j-w[i]] + v[i])$$

“将前 i 件物品放入容量为 j 的背包中”这个子问题，若只考虑第 i 件物品的策略（放或不放），

那么就可以转化为一个只牵扯前 i−1 件物品的问题。

如果不放第 i 件物品，那么问题就转化为“前 i−1 件物品放入容量为 j 的背包中”，价值为 f[i−

1][j];

如果放第 i 件物品，那么问题就转化为“前 i−1 件物品放入剩下的容量为 j−c[i]的背包中”，此

时能获得的最大价值就是 f[i−1][j−w[i]],再加上通过放入第 i 件物品获得的价值 v[i]。

因此得出上面的式子。
继续优化空间（利用之前提到的知识）：
如果我们压缩到一维空间解题，这次我们需要的是上面的位置和左上的位置，也就是说，我们需要左边的位置是没被更新过的，得出更新顺序应该从右往左：

```
for i in range(1,n+1):
    for j in range(v,-1,-1)
            f[j] = max(f[j], f[j - w[i]] + v[i]);
```

3.2 完全背包

有N种物品和一个容量为V的背包，每种物品都有无限件可用。第i种物品的费用是$w[i]$，价值是$v[i]$。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

这个问题非常类似于 01 背包问题，所不同的是每种物品有无限件。也就是从每种物品的角度考虑，与它相关的策略已并非取或不取两种，而是有取 0 件、取 1 件、取 2 件……等很多种。如果仍然按照解 01 背包时的思路,很容易得出：

$$f[i][j] = max(f[i-1][j-k*w[i]]+k*v[i])|0 <= k*w[i] <= j$$

这跟 01 背包问题一样有 O(VN)个状态需要求解，但求解每个状态的时间已经不是常数了

而是 $O(V/w[i])$ ，总的复杂度可以认为是 $(N*\Sigma(V/w[i]))$,将 01 背包问题的基本思路加以改进，得到了这样一个清晰的方法。这说明 01 背包问题的方程的确是很重要，可以推及其它类型的背包问题。但我们还是试图改进这个复杂度。
我们可以知道，对于一个普遍位置 w，当前物品代价为 2 的话，下图中红色区域就是和位置 w 的取值相关的一些数值：

	A	B	C	D	E	F	G
1							
2	3		2		1		
3							w

对当前物品的决策就依次是：不拿、拿一个、拿两个、拿三个（对应上面式子中的 k）
我们算法优化的思路就是不断去除重复计算，显然我们可以继续优化这个式子。
请思考：我们的 E3 位置是如何得出的？其实是根据三个红色区域得出的，但是我们算位置 w 时又算了一遍，显然是重复了。而 E3 其实包含了不拿、拿一个、拿两个这些情况中的最优解，我们算 w 时直接用就可以了。

给出模板代码：

```
for (int i = 1; i <= n; i++)
```

```
for (int j = w[i]; j <= V; j++)
        f[j] = max(f[j], f[j - w[i]] + v[i]);
```

对比两种背包:
这个代码与 01 背包的代码只有 j 的循环次序不同而已。为什么这样一改就可行呢?
首先想想为什么 01 背包中要按照 j=V...0 j=V...0j=V...0 的逆序来循环。这是因为要保证第 i 次循环中的状态 f[i][j]是由状态 f[i−1][j−w[i]]递推而来。换句话说，这正是为了保证每件物品只选一次，保证在考虑“选入第 i 件物品”这件策略时，依据的是一个绝无已经选入第 i 件物品的子结果 f[i−1][j−w[i]]。
而现在完全背包的特点恰是每种物品可选无限件，所以在考虑“加选一件第 i ii 种物品”这种策略时，却正需要一个可能已选入第 i 种物品的子结果 f[i][j−w[i]]，所以就可以并且必须采用 j=0...V j=0...Vj=0...V 的顺序循环。这就是这个简单的程序为何成立的道理。
最终给出状态转移方程给不明白的同学看:

$$f[i][j] = max(f[i-1][j], f[i][j-w[i]] + v[i])$$

（也可以通过数学导出此式）

3.3 多重背包

> 有N种物品和一个容量为V的背包。第i种物品最多有$p[i]$件可用，每件费用是$w[i]$，价值是$v[i]$。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

和之前的背包不同，每种物品不是只有一件，也不是有无限件，这次的每种物品的数量都是有限制的，我们对于每种物品，可以选择拿一件、两件……p[i]件。
我们借用上一种问题的图:

	A	B	C	D	E	F	G
1							
2	3		2		1		
3							w

看起来是类似的，位置 w 依旧和红色区域相关，但是我们可以直接根据 E3 来求出位置 w 吗?是不能的，因为条件变了，每种物品不是无限的，可能在 w 位置，图中椭圆圈出的位置代表着需要拿三个，但是如果规定最多拿两个，我们这种算法就出问题了。

一种做题思路: 把每个物品都按 01 背包做: 比如第 i 种物品，我们就按有 p[i]件相同的物品。每一种物品都是如此，按 01 背包做就可以了。（但是显然很蠢）

改进:
我们平时买东西时，难道带的全是一元的硬币吗？当然不是，只要手中的钱可以凑出商品的价格即可，比如 9 元的东西，我不一定用九个硬币（背包问题的物品）来付钱，可以 5 元+4 个 1 元。

背包问题也一样，我们不一定要全部拆成 1 的物品，只要我们的物品可以代表 0——>p[i]的所有情况，我们就认为这种策略是正确的。
那如何拆 p[i]个物品可以保证我们的物品可以代表 0——>p[i]的所有情况呢？这里要借助 2 进制思想。
一个 n 位的二进制数可以取 0 到 2 的 n 次方-1，第 i 位代表的是 2 的 i-1 次方。
对应到物品：
我们的 p[i]=15，我们怎样拆呢？
1+2+4+8 即可，这四个数一定可以组合出 0-15 的任何一个数。

二进制拆分代码如下：

```
for (int i = 1; i <= n; i++) {
        int num = min(p[i], V / w[i]);
        for (int k = 1; num > 0; k <<= 1) {
                if (k > num) k = num;
                num -= k;
                for (int j = V; j >= w[i] * k; j--)
                        f[j] = max(f[j], f[j - w[i] * k] + v[i] * k);
        }
}
```

3.4 一些变形选讲

1）最常见的一些变形，甚至不能说是变形，上面也提到过，但是怕同学们不知道：
我们常见的问题中，一般是问最优解，可能是最大，或者最小，但是，问题也可能是方法的数量，这个时候，一般把状态转移方程中的 max(min)改为 sum（求和）即可，当然，压缩空间后的样子还是需要自己写。
2）初始化的细节问题
我们看到的求最优解的背包问题题目中，事实上有两种不太相同的问法。有的题目要求"恰好装满背包"时的最优解，有的题目则并没有要求必须把背包装满。这两种问法的区别是在初始化的时候有所不同。
如果是第一种问法，要求恰好装满背包，那么在初始化时除了 f[0]为 0 其它 f[1...V]均设为

$-\infty$ $-\infty$$-\infty$，这样就可以保证最终得到的 f[N]是一种恰好装满背包的最优解。

如果并没有要求必须把背包装满，而是只希望价格尽量大，初始化时应该将 f[0...V]全部设为 0。
为什么呢？可以这样理解：初始化的 f 数组事实上就是在没有任何物品可以放入背包时的合法状态。如果要求背包恰好装满，那么此时只有容量为 0 的背包可能被价值为 0 的 nothing "恰好装满"，其它容量的背包均没有合法的解，属于未定义的状态，它们的值就都应该是

$-\infty$了。如果背包并非必须被装满，那么任何容量的背包都有一个合法解"什么都不装"，这个解的价值为 0，所以初始时状态的值也就全部为 0 了。
3）常数优化
前面的代码中有 for(j=V...w[i])，还可以将这个循环的下限进行改进。

由于只需要最后 f[j]的值，倒推前一个物品，其实只要知道 f[j−w[n]]即可。以此类推，对以

第 j 个背包，其实只需要知道到 f[j−sumw[j…n]]即可，代码自行修改。

4）其实拆解二进制物品并不是多重背包的最优解，但是最优的单调队列思想写起来有些繁琐，可能以后会写。