dynamic Programming

动态规划是20世纪50年代由Richard Bellman发明的。不像贪婪算法，回朔算法等，单从名字上根本理解这是什么鬼。Bellman自己也说了，这个名字完全是为了申请经费搞出来的，所以说这个名字坑了一代又一代的人。

dynamic Programming，是一种高效解决问题的方法，使用与具有**重复子问题**和**最优子结构**的问题。

如果可以把局部子问题的解结合起来得到全局最优解，那么这个问题就具备**最优子结构**。

如果计算最优解时需要处理很多相同的问题，那么这个问题就具备**重复子问题**。

# Fibonacci sequence（斐波那契数列）

fibonacci数列是递归算法的一个典型的例子，这里不介绍了，大家都懂，直接上代码：

import time

def Fibnacci(n):

if n==0 or n==1:

return 1

else:

return Fibnacci(n-1) + Fibnacci(n-2)

num=37

start = time.clock()

Fibnacci(num)

end = time.clock()

print "Fibnacci sequense costs",end-start

结果耗时：

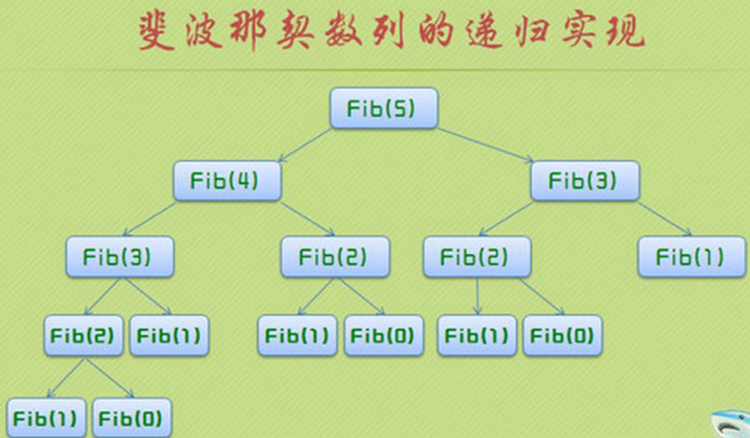
Fibnacci sequense costs 14.9839106433

这速度也是醉了啊，这才是37位啊，手算也就2分钟的事啊。

如果试下Fibnacci(120) ，这个千万不敢试，会怀孕，说错了，是会看不到明天的太阳。（官方统计算完基本上要250000年）

那么为什么会这么慢呢。我们以Fibnacci(5)位例子，每次计算Fibnacci(5)都要计算Fibnacci(4)和Fibnacci(3)，而Fibnacci(4)要计算Fibnacci(3)和Fibnacci(2)，Fibnacci(3)要计算Fibnacci(2)和Fibnacci(1)，一直这样递归下去，作了大量的重复计算。而函数调用时很费时间和空间的。

有一个图很好的说明了这个问题：



既然重复计算如此耗时，那么能不能不重复计算这些值呢？当第一次计算了这些值的时候，我们把他们缓存起来，等到再次使用的时候，直接把他们拿过来用，这样就不用做大量的重复计算了。这就是动态规划的核心思想。

还是以Fibnacci为例子：

每当我们第一次计算Fibnacci(n)的时候，我们就将其缓存到memo的列表中，当需要再次计算Fibnacci(n)的时候，首先去memo的列表中查找，如果找到了就直接拿来用，找不到再计算。下面是具体的程序：

def fastFib(n,memo={}):

if n==0 or n==1:

return 1

try:

return memo[n]

except KeyError:

result = fastFib(n-1,memo) + fastFib(n-2,memo)

memo[n] = result

return result

实验下效果：

n=120

start=time.clock()

fastFib(n)

end=time.clock()

print"Fibnacci sequense costs",end-start

Fibnacci sequense costs 0.00041543479823

这就是差距啊！从算法复杂度上来讲这次的算法每次只调用fastFib函数一次，所以复杂度为O(n)。**这就是差距的原因。**