**Memoization** is a term describing an optimization technique where you cache previously computed results, and return the cached result when the same computation is needed again.

**Dynamic programming** is a technique for solving problems recursively and is applicable when the computations of the subproblems overlap.

Dynamic programming is typically implemented using tabulation, but can also be implemented using memoization. neither one is a "subset" of the other.

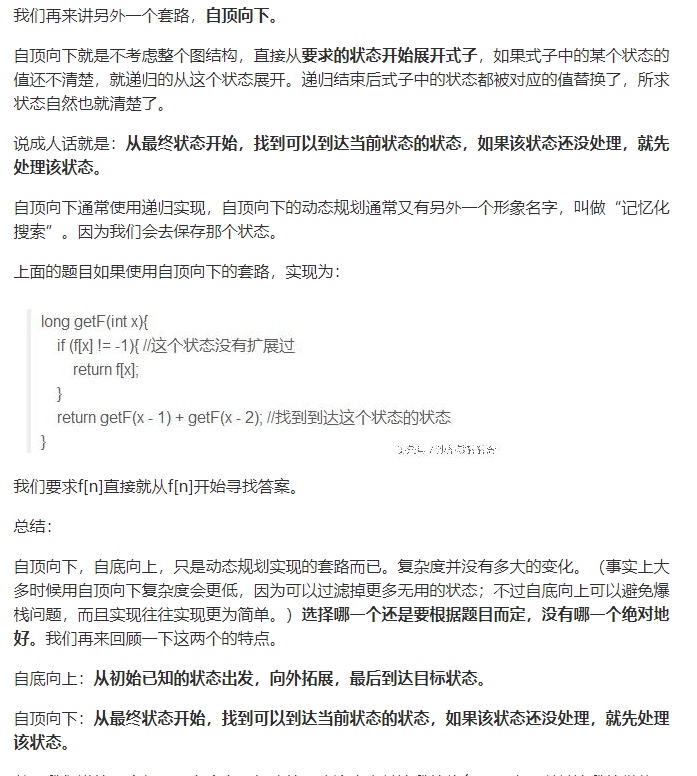
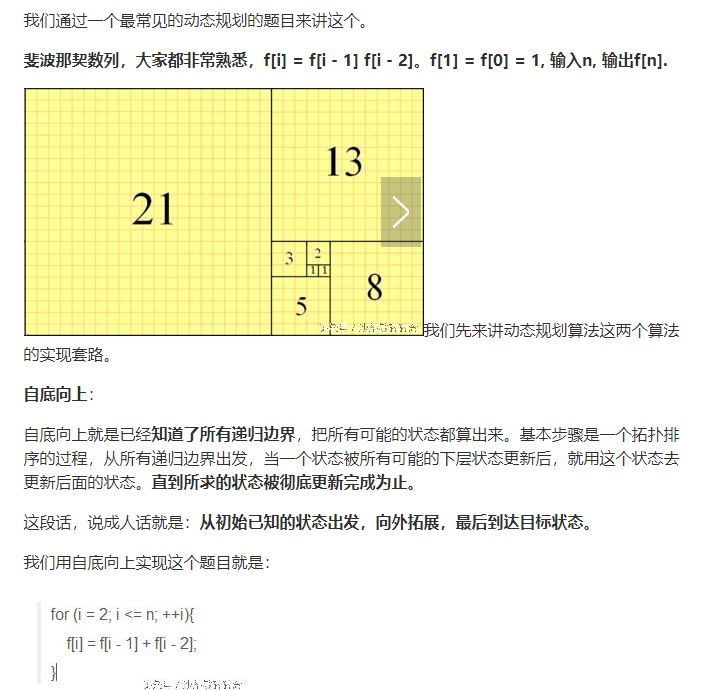
A reasonable follow-up question is: **What is the difference between tabulation (the typical dynamic programming technique) and memoization?**

When you solve a dynamic programming problem using tabulation you solve the problem "**bottom up**", i.e., by solving all related sub-problems first, typically by filling up an *n*-dimensional table. Based on the results in the table, the solution to the "top" / original problem is then computed.

If you use memoization to solve the problem you do it by maintaining a map of already solved sub problems. You do it "**top down**" in the sense that you solve the "top" problem first (which typically recurses down to solve the sub-problems).

A good slide from [here](http://web2.cc.nctu.edu.tw/~claven/course/Algorithm/unit13.ppt) (link is now dead, slide is still good though):

* If all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms a top-down memoized algorithm by a constant factor
  + No overhead for recursion and less overhead for maintaining table
  + There are some problems for which the regular pattern of table accesses in the dynamic-programming algorithm can be exploited to reduce the time or space requirements even further
* If some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required



满足三个条件之一：

Maximum/minimum

Yes/no

Count(\*)

（附加条件）can not swap

列出所有方案的，一定不是动态规划（用递归）

4要素：

State

Function

Initialization

answer

4类型：

Matrix DP 0.1

Sequence 0.4

Two sequence 0.4

Backpack 0.1

不常用递归解决，为什么（递归有额外的堆栈的开销）

矩阵的初始化要考虑两边

对于sequence ，实现时往往将数组大小设置为n.length + 1，留出F[0]。（前0的空串不能忽略）

\*在实现DP时，注意适当进行预处理，避免时间复杂度从n^2 变成n^3

\*单词 和 字符串 有区别 （单词有长度，最长也就20+，字符串长度任意）

节约空间：

只和i – 1 有关系就% 2 ， 和 I – 2 也有关系就 % 3；