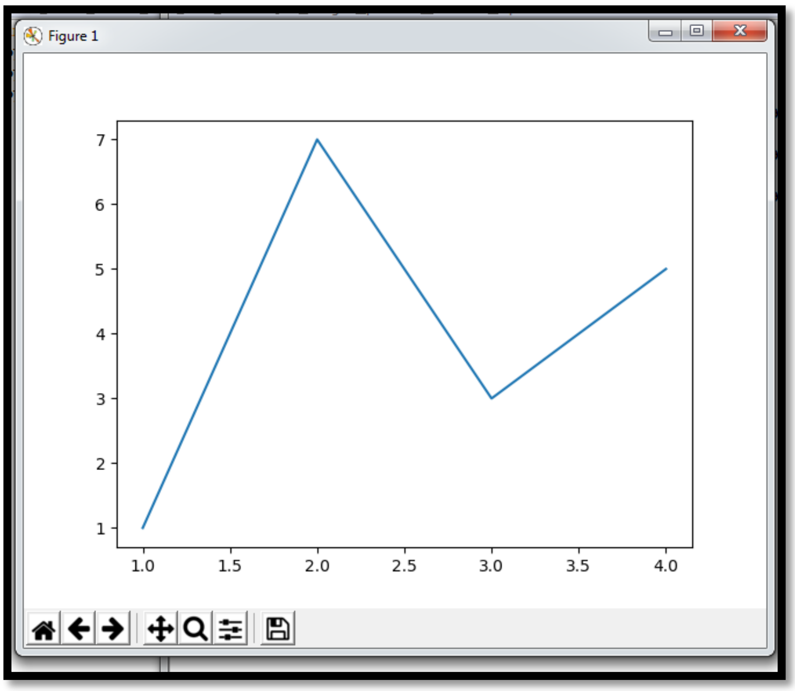
Plotting Using PyLab

- PyLab is a Python standard library module that provides many of the facilities of MATLAB, "a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numeric computation.
- A complete user's guide for PyLab is at the website *matplotlib.sourceforge.net/users/index.html*.
- Simple Example that uses pylab.plot:

```
import pylab
pylab.figure(1)
pylab.plot([1,2,3,4],[1,7,3,5])
pylab.show()
```

- Following output will be displayed while executing above script. The exact appearance may depend on the operating system of computer system.



- The bar at the top contains the name of the window (Eg: "Figure 1").
- The middle section of the window contains the plot generated by the invocation of *pylab.plot.*

- The two parameters of *pylab.plot* must be sequences of the same length. The first specifies the x – coordinates of the points to be plotted, and the second specifies the y – coordinates. That is, <x , y> coordinate pairs.
- As each point is plotted, a line is drawn connecting it to the previous point.
- The final line of code – *pylab.show()*, causes the window to appear on the computer screen. If this line is omitted, the figure would still have been produced, but it would not have been displayed.
- The bottom bar of the window contains a number of push buttons:
    - The rightmost button is used save figure at specified location with the extension .png (Portable Networks Graphics Format).
    - The next button to the left is used to adjust the appearance of the plot in the window.
    - The next four buttons are used for panning and zooming.
    - The button on the left is used to restore the figure to its original appearance after you are done playing with pan and zoom.
- It is possible to produce multiple figures and write them to files.

```
import pylab
pylab.figure(1) #create figure 1
pylab.plot([1,2,3,4], [1,2,3,4]) #draw on figure 1

pylab.figure(2) #create figure 2
pylab.plot([1,4,2,3], [5,6,7,8]) #draw on figure 2
pylab.savefig('img1') #save img1

pylab.figure(1) #go back to working on figure 1
pylab.plot([5,6,10,3]) #draw again on figure 1
pylab.savefig('img2') #save img2
```

- The above code produces multiple figures and saves to files img1.png and img2.png.
- PyLab has a notion (concept) of "current figure." Executing *pylab.figure(x)* sets the current figure to the figure numbered *x*. Subsequently executed calls of plotting

functions implicitly refer to that figure until another invocation of *pylab.figure* occurs.

- We can have all plots with informative titles, all axes with label and different formatting styles (e.g. line color, line style, font, font color, font style etc.). – [*See Programs for further detail]*

Plotting Mortgages, an Extended Example

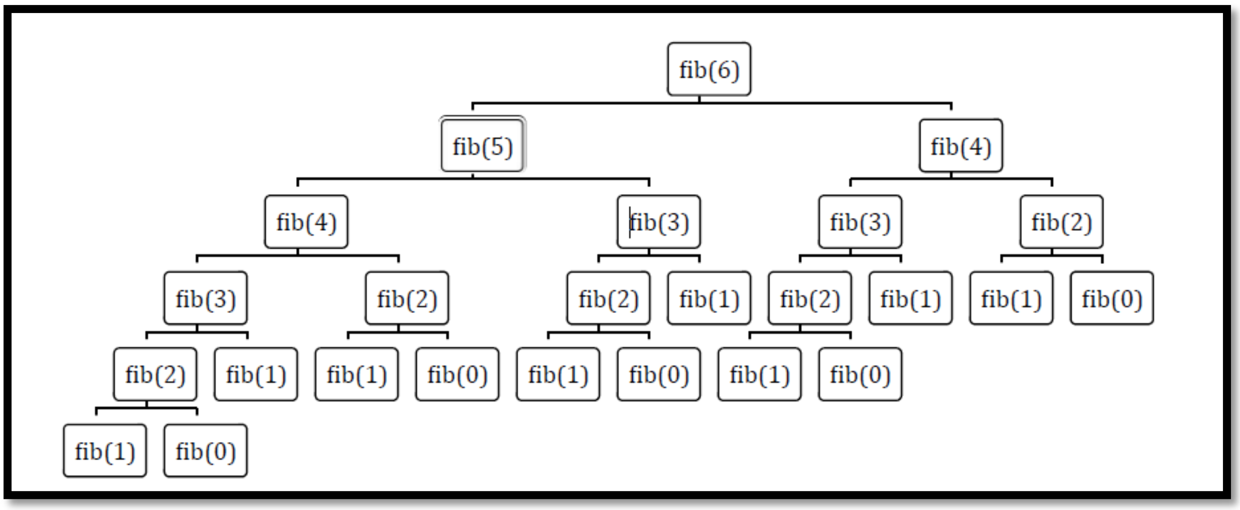
- *See Program for Mortgages with different Plottings…*

Fibonacci Sequence Revisited

- Previously, we have following recursive implementation of the Fibonacci function.

```
Example:
    >>> def fib(n):
              if n == 0 or n==1:
                   return 1
              else:
                   return fib(n-1) + fib(n-2)
```

- While this implementation of the recurrence (return) is obviously correct, it is terribly inefficient.
- Check for fib(120), its running time is substantial (large) / takes time to execute. We cannot wait for it to complete.
- The complexity of the implementation is a bit hard to derive, but it is roughly O(fib(n)). That is, its growth is proportional to the growth in the value of the result, and the growth rate of the Fibonacci sequence is substantial. For example, fib(120) is 8,670,007,398,507,948,658,051,921. If each recursive call took a nanosecond, fib(120) would take about 250,000 years to finish.
- Now, Why this implementation takes so long. Given the tiny amount of code in the body of fib, it’s clear that the problem must be the number of times that fib calls itself. As an example, look at the tree of calls associated with the invocation fib(6).



Tree calls for recursive Fibonacci

- Notice that we are computing the same values over and over again. For example fib gets called with 3 three times, and each of these calls provokes four additional calls of fib.
- It might be a good idea to record the value returned by the first call, and then look it up rather than compute it each time it is needed. This is called *memorization*, and is the key idea behind dynamic programming.

Example:

```
>>> def Fib(n, memo = {}):
          if n == 0 or n==1:
               return 1
          try:
               return memo[n]
          except KeyError:
               res = Fib(n-1,memo) + Fib(n-2),memo)
               memo[n] = res
               return res
```

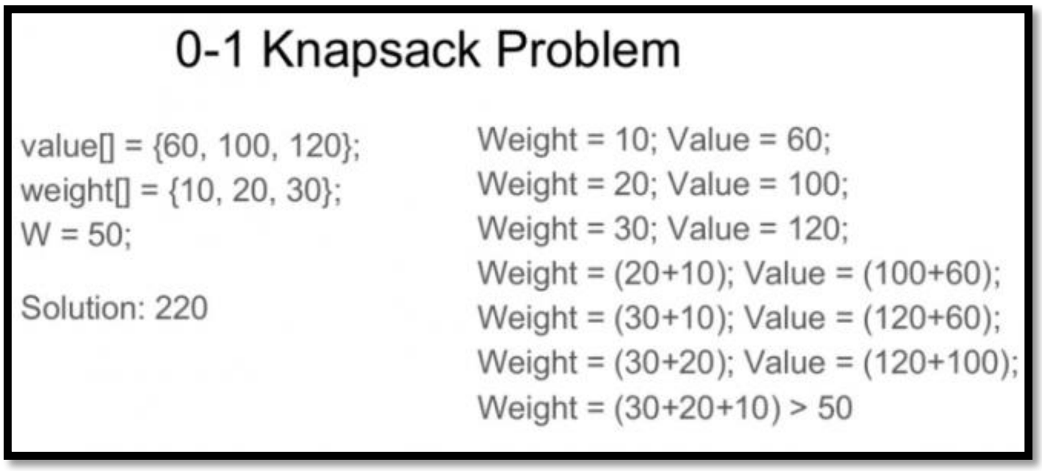
- Above code is with an implementation of Fibonacci based on this idea.
- The function *Fib* has a parameter, memo, that it uses to keep track of the numbers it has already evaluated.
- The parameter has a default value, the empty dictionary, so that clients of *Fib* don't have to worry about supplying an initial value for memo.

- When *Fib* is called with an n > 1, it attempts to look up n in memo. If it is not there (because this is the first time *Fib* has been called with that value), an exception is raised.
- When this happens, *Fib* uses the normal Fibonacci recurrence, and then stores the result in memo.
- After implementation of above *Fib()* function, we will see that it is indeed quite fast: fib(120) returns almost instantly.
- For each value from 0 to n, *fib* is called exactly once. So, the time complexity of *Fib* is O(n).

Dynamic Programming and the 0/1 Knapsack Algorithm:

- Dynamic Programming was invented by Richard Bellman in the early 1950s.
- Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again.
- A problem has *optimal substructure* if a globally optimal solution can be found by combining optimal solutions to local sub problems. For Example, Merge Sort which exploits (uses) that a list can be sorted by first sorting sub lists and then merging the solutions.
- A problem has *overlapping sub problems* if an optimal solution involves solving the same problem multiple times. Merge sort does not exhibit (show) this property. Even though we are performing a merge many times, we are merging different lists each time. Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to be recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again. For example, Binary Search doesn't have common subproblems.
- The 0/1 knapsack problem exhibits both of above properties.

- 0/1 Knapsack Problem:
    - Knapsack problem is also called as *rucksack problem.*
    - In 0/1 Knapsack problem, items can be entirely accepted or rejected.
    - Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0 .. n-1] and wt[0 .. n-1] which represent values and weights associated with n items respectively. Find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. We cannot break an item, either pick the complete item, or don't pick it (0 / 1 property).



0-1 Knapsack Problem

value[] = {60, 100, 120};
weight[] = {10, 20, 30};
W = 50;

Solution: 220

Weight = 10; Value = 60;
Weight = 20; Value = 100;
Weight = 30; Value = 120;
Weight = (20+10); Value = (100+60);
Weight = (30+10); Value = (120+60);
Weight = (30+20); Value = (120+100);
Weight = (30+20+10) > 50

    - A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.
        - Optimal Substructure: To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.

          Therefore, the maximum value that can be obtained from n items is max of following two values: 1) Maximum value obtained by n-1 items and W weight (excluding nth item). 2) Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

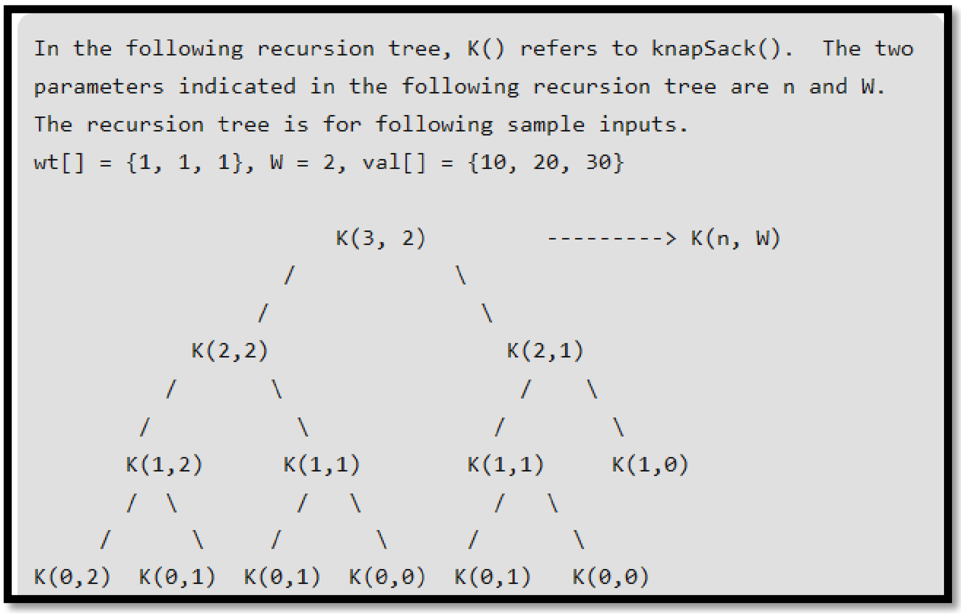
- Overlapping Sub problems: Following is recursive implementation that simply follows the recursive structure mentioned above.

```
def test(W , wt, val , n):
     if n == 0 or W == 0 :
          return 0

     if (wt[n-1] > W):
          return test(W , wt , val , n-1)
     else:
        return  max(val[n-1] + test(W-wt[n-1] ,
     wt , val , n-1), test(W , wt , val , n-1))

⇨ To Test above function:
  >>> val = [60, 100, 120]
  >>> wt = [10, 20, 30]
  >>> W = 50
  >>> n = len(val)
  >>> print (test(W, wt, val, n))
  220     #Output
```

- It should be noted that the above function computes the same subproblems again and again. See the following recursion tree, K(1, 1) is being evaluated twice. Time complexity of this naive recursive solution is exponential (2^n).



```
In the following recursion tree, K() refers to knapSack().  The two
parameters indicated in the following recursion tree are n and W.
The recursion tree is for following sample inputs.
wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}

                       K(3, 2)         ---------> K(n, W)
                   /            \
                 /                \
            K(2,2)                  K(2,1)
          /       \                 /    \
        /           \             /        \
       K(1,2)      K(1,1)        K(1,1)     K(1,0)
       /  \         /   \          /   \
     /      \     /       \      /       \
K(0,2)  K(0,1)  K(0,1)  K(0,0)  K(0,1)   K(0,0)
```

- Since sub problems are evaluated again, this problem has Overlapping Sub problems property. So the 0-1 Knapsack problem has both properties of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same sub problems can be avoided by constructing a temporary array K[][] in bottom up manner. Following is Dynamic Programming based implementation.

```
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W+1)] for x in range(n+1)]

    for i in range(n+1):
        for w in range(W+1):
            if i==0 or w==0:
                K[i][w] = 0
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1] + K[i-1]
                    [w-wt[i-1]],  K[i-1][w])
            else:
                K[i][w] = K[i-1][w]

    return K[n][W]
```

Dynamic Programming and Divide-and-Conquer:

- Like divide-and-conquer algorithms, dynamic programming is based upon solving independent sub problems and then combining those solutions. There are, however, some important differences:
- Divide-and-conquer algorithms are based upon finding sub problems that are substantially smaller than the original problem. For example, merge sort works by dividing the problem size in half at each step. In contrast, dynamic programming involves solving problems that are only slightly smaller than the original problem. For

example, computing the 19th Fibonacci number is not a substantially smaller problem than computing the 20th Fibonacci number.

- Another important distinction is that the efficiency of divide-and-conquer algorithms does not depend upon structuring the algorithm so that the same problems are solved repeatedly. In contrast, dynamic programming is efficient only when the number of distinct sub problems is significantly smaller than the total number of sub problems.
- So, at the end, the Divide and Conquer should be used when same sub problems are not evaluated many times. Otherwise Dynamic Programming / Memoization should be used. For example, Binary Search is a Divide and Conquer algorithm, we never evaluate the same sub problems again; whereas for calculating nth Fibonacci number, Dynamic Programming should be preferred.