MEMOIZATION TECHNIQUE
(For speeding up a brute-force code)

Start with the brute-force program for rod cutting problem (from the last class)

```
import sys
sys.setrecursionlimit(10000)

p = input().split()
L = len(p)
for i in range(L):
    p[i] = int(p[i])

p.insert(0,0)

def maxRev(l):
    global p,L

    if l == 0:
        return 0
    else:
        mx = 0
        for i in range(1,l+1):
            mx = max(mx, p[i]+maxRev(l-i))
    return mx

print(maxRev(L))
```

1. Create a list named "calls" with size equal to 1 + the largest possible rod length. Initialize all the values in the list to 0. (This requires only basic Python knowledge)

2. Modify the brute-force solution so that the program collects the total number of recursive calls, i.e. increment calls[l] for every call to $\mathrm{maxRev}(l), 1 \le l \le L$. Observe how many times each $\mathrm{maxRev}()$ is called by examining the final values in "calls" list.

3. For *each* value of l, does $\mathrm{maxRev}(l)$ return the same or different values ?

4. Therefore, for length of L, at most how many calls to all $\mathrm{maxRev}()$'s should be adequate ?

5. What is the reason that there were too many calls to $\mathrm{maxRev}()$ in the brute-force solution ?

6. Suggest a modification to the brute-force solution such that once a value of an $\mathrm{maxRev}(l)$ is already computed, it will never have to be recomputed again. Only idea is needed in this step.

7. Implement the modified solution.

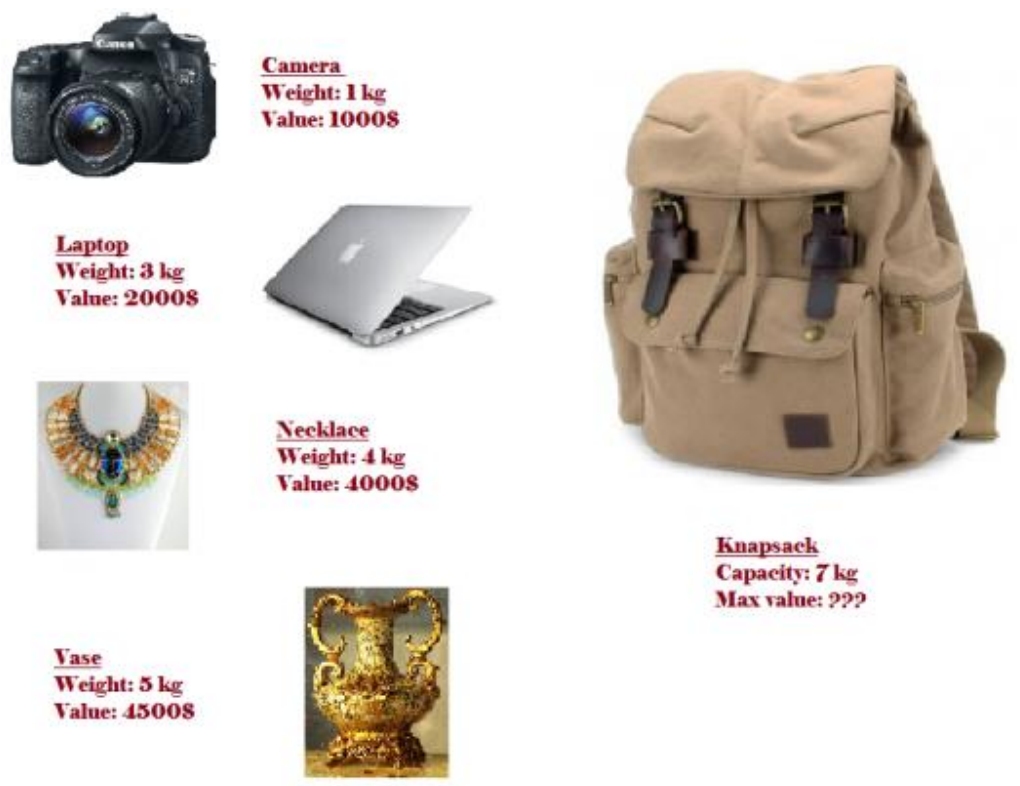
8. Observe how faster the algorithm becomes (comparing the total number of recursive calls).

9. Apply the same concept to speed up the minimum coin change problem. (last class)

```
import sys
sys.setrecursionlimit(10001)

c = input().split()
for i in range(len(c)):
    c[i] = int(c[i])

V = int(input())

def mincoin(v):
    global c

    if v == 0:
        return 0
    else:
        minc = 10000000000
        for x in c:
            if x <= v:
                minc = min(minc, 1 + mincoin(v-x))
        return minc

print(mincoin(V))
```

10. If done correctly, the memoized version (of solution for minimum coin change) will be able to handle the 2nd test case in the example.

PROBLEM B: 0-1 KNAPSACK



Given a maximum weight you can carry in a knapsack and items, each with a weight and a value, find a set of items you can carry in the knapsack so as to maximize the total value.

Version 1: BRUTE-FORCE

1. The code below applies the brute-force combination technique to solve this problem. Every complete combination returns the total value if capacity is not exceeded, or returns -1 if exceeded.

```
import sys
sys.setrecursionlimit(10000)

N,M = map(int, input().split())
w = list(map(int, input().split()))
v = list(map(int, input().split()))

x = [0]*N

def comb(i):   # considering item i
    if i == N:
        sw = sv = 0
        for j in range(N):
            if x[j] == 1:
                sw += w[j]
                sv += v[j]
        if sw > M:
            return -1
        else:
            return sv
    else:
        x[i] = 0
        a = comb(i+1)
        x[i] = 1
        b = comb(i+1)
        return max(a,b)

print(comb(0))
```

2. **ISSUE:** Based on technique in step 1 above, suppose that items are determined in order from item 0 to n-1, when the algorithm *is deciding* between selecting item i or not, there is no associated information of how the items 0 to i-1 have been selected!
   Therefore, the total number of states that decide on selecting item i is *the total number of ways to select items* 0 *to* i-1,

   which is ________________________

   - Accordingly, the answers of selecting item i may result in different values, yes or no?
   - Consequently, can we memoize this brute-force code for speed-up?

Version 2: BRUTE_FORCE WITH TWO STATE VARIABLES

3. However, suppose that when deciding between selecting item i or not, the *currently* available capacity of the knapsack is also specified. Then the possible items to be added to the knapsack will become more constrained. For example, if the currently available capacity is 0, we know that it is impossible to add any more item.

   In this way, a state of this problem can be defined with two components.

   - the current item being considered
   - the currently available capacity of the knapsack (indicating that some of items 0 to i-1 has taken some space in the knapsack)

```
N,M = map(int, input().split())
w = list(map(int, input().split()))
v = list(map(int, input().split()))

def maxVal(i,C):         # index i, capacity C
    if i == N:
        return 0
    else:
        skip = maxVal(i+1,C)
        if w[i] <= C:    # w[i] does not exceed capacity
            take = v[i] + maxVal(i+1,C-w[i])
        else:
            take = -1
        return max(skip, take)

print(maxVal(0,M))
```

4. Add a part of the code for counting the total number of recursions. Print the number of recursive calls as an output of the program. Download the test cases from Class Materials. Then test run. Note the case where the running time is excessively long.

Version 2: MEMOIZATION

5. Observe. Is each state generated by the brute-force algorithm unique? In other words, are there repeated occurrences of the same state ?

6. Given a problem state, does any recursive call on this state return the same value? Why?

7. If your answer to question 5 above is “Yes”, memoization technique can minimize the number of repeated recursive calls. Modify the code into memoized version.

8. Compare the total number of recursive calls against the brute-force version.