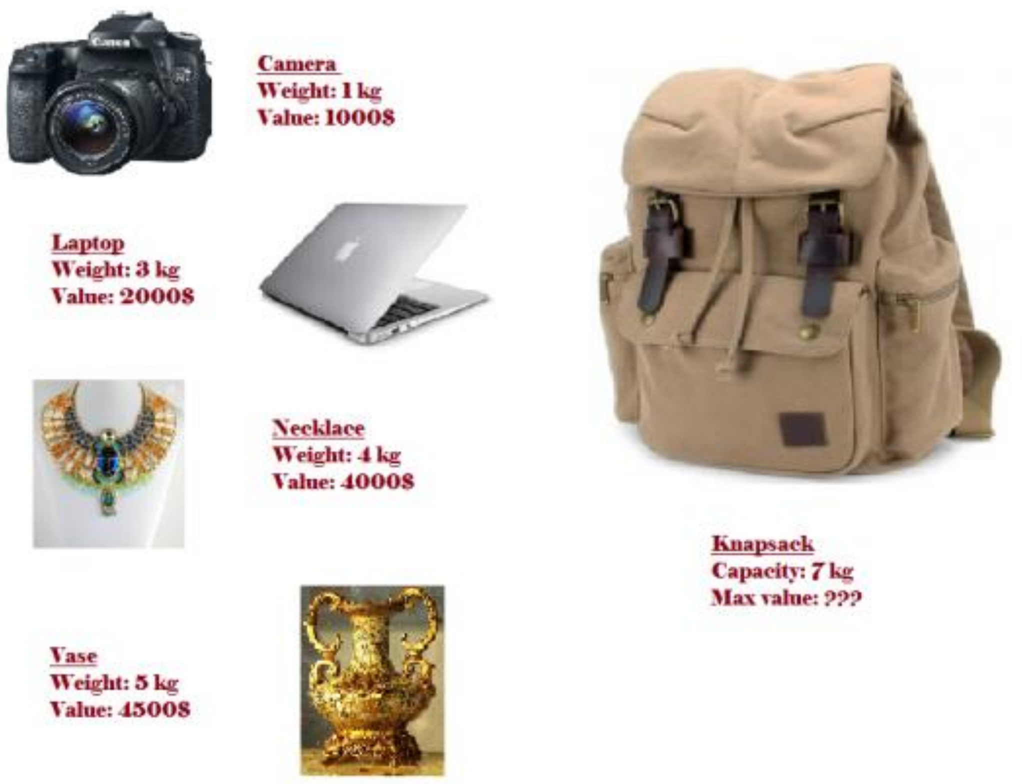
PROBLEM A: 0-1 KNAPSACK



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.

PROBLEM B: EDIT DISTANCE

The edit distance between two words—sometimes also called the *Levenshtein* distance—is the *minimum* number of letter insertions, letter deletions, and letter substitutions required to transform one word into another.

For example, the edit distance between FOOD and MONEY is at most four:

FOOD → MOOD → MON_D → MONED → MONEY

Given two strings, find the edit distance between them.

INPUT:

Line 1: the first string, A

Line 2: the second string, B

OUTPUT:

Edit distance between the two strings

1) We are transforming string A to string B. Assume that string A[0] .. A[i-1] have been transformed to be identical to B[0] .. B[j-1], and the consideration now is on A[i] and B[j].

   The table below lists all possible scenarios at state (i, j) and edit operations that can be performed. What is the consequential state for each combination of condition and operation ?

condition	edit operation	next state to consider
A[i] == B[j]	None	
A[i] != B[j]	Insert B[j] in front of A[i]	
A[i] != B[j]	Delete A[i]	
A[i] != B[j]	Change A[i] to B[j]	

2) What is the beginning state?

3) If A runs out, but B has not yet, in other words, i == len(A), but j < len(B), what is the additional edit distance required to complete the transformation?

4) If B runs out, but A has not yet, what is the additional edit distance required to complete the transformation?

5) Use the concepts obtained from step 1 to 4 above in write a recursive brute-force solution for this problem. The zipped test case file is downloadable from Class Materials.

6) Given that a string can be up to 1000 letters long, improve the brute-force solution so that the program will finish in no more than 2.5 seconds (CPU processing time).