# Assignment 5: Solve the 0-1 Knapsack Problem using Brute Force

## Learning Outcomes

* Learn a brute force approach to a difficult problem.

## Motivation

Brute Force is considered the slowest and least efficient of all of the programming techniques that we will cover in this class. Still, it's a useful technique for when you aren't sure how to solve a problem since it should always give you the best results every time, whereas there aren't the same guarantees with other techniques. It's nice to have this approach for when you are testing a (hopefully) must better approach (and there is almost always a better approach).

## Inputs

The story of the 0-1 knapsack problem is that you are a robber and must rob a store with the greatest total value of the items in knapsack as possible without exceeding the weight limit of your knapsack.

* The first integer seen in the input will be the weight limit of your knapsack.
* The second integer seen will be the number of items in the fictional store which we are attempting to rob.
* Every input after that will be a product represented by three values.
  + A single word string representing the name of item, such as "bike" or "toycar". Notice that the name will not have any spaces in it.
  + An integer representing the weight of the item.
  + An integer representing the value of the item.

## Outputs

* Your program should report all of the items which you put into your knapsack. It should have the maximum value without exceeding the weight limit of the knapsack.
* You should report the total value.
* You should report the total weight.

## How to solve the problem.

This algorithm is recrusive. Can you determine the runtime complexity?

1. Inputs: Weight limit remaining in pack, All items currently in knapsack, All items in the store (which is constant), an index position for the item you are currently viewing.
   1. If the index is greater than or equal to the size of the products, then return knapsack
   2. Get a version of the future knapsack in which you do not pick up the item at position index. This is a recursive call.
   3. If you are able to add the item at the position index to the knapsack, then
   4. Add that item to the knapsack
   5. Get a version of the future knapsack in which you do pick up the item at position index. This is a recursive call.
   6. Compare your two knapsacks by total value.
   7. Return the knapsack with the larger value.

## Examples.

Example 1.

5  
3  
speaker 2 100  
bike 3 120  
nicechair 4 1000

Results.

Name: nicechair Weight: 4 Value: 1000  
  
Total Weight: 4  
Total Value : 1000

Example 2.

5  
3  
speaker 2 100  
bike 3 120  
chair 4 200

Results.

Name: speaker Weight: 2 Value: 100  
Name: bike Weight: 3 Value: 120  
  
Total Weight: 5  
Total Value : 220

## Turn it in

Upload a zip file containing the files that use used or wrote yourself to the drop box on D2L:

* Make sure your name, CSCI 4270, and the Programming Assignment Number appear in comments in all of your files.
* Note: NO CREDIT will be given to programming assignments that do not compile.
* Make sure you have compiled and tested your program before handing it in.