**Introduction:**

In Lab 7 we were tasked to implement several version of solutions of the knapsack problem, basically given list of weights and values of an object as well as debt and weight the knapsack can carry is it possible to pay the debt with the allotted weight. We implement many different versions to solve the same problem, a backtracking solution, a greedy solution, and random solution

**Proposed Solution Design and Implementation:**

Method Knapsack\_backtracking(W,D,v,w):

This is the original method we were tasked in implementing and is the simplest. Given a weight and debt and list representing values and weights of objects it will determine if its possible to get rid of the debt with the Weight allotted. First we have a basecase to check if the weight is less then 0 if it is we return false then another for if the debt is less then 0 then we return true, and the last base case if either v or w is empty then we also return false. Otherwise we return the method with weight minus the first element in the weight list, the debt – the first element in the value list, and the slice of the value and weight list starting at 1 or we return the method with the same weight and debt and the slice of the weight and value list starting at index 1. This method although simple runs in O(2^n)

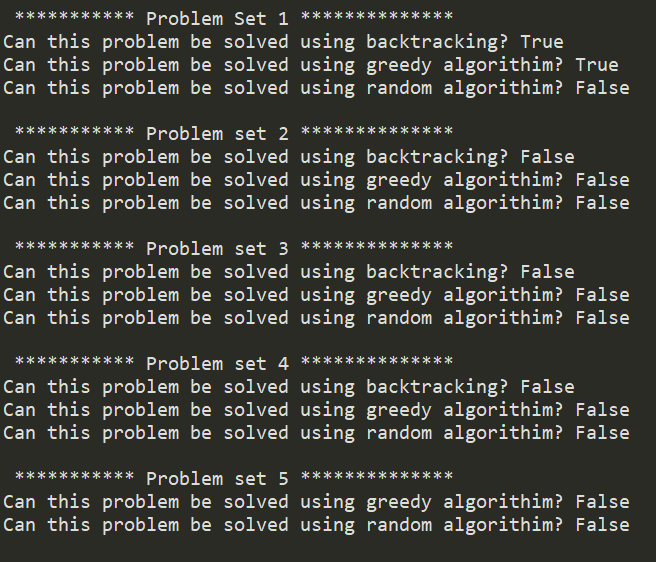
Method Kanpsack\_Greedy(W,D,v,w):

This method is solves the exact same problem as the one above it, but it does so in a greedy approach. First I convert the list v and w in np arrays to make use of some of the functions that NumPy has, then I create a list that is filled with the weight to value ratios of all the items this can be done easier by just dividing the two NumPy arrays. Then we create a list called index that is the indexes of the greatest value to weigh ration items we get this list by using the np.argsort function on the wvr in descending order once we have the indexes we can just iterate for the length of v or w and subtract the weight and value from the item at index[i] if the weight is equal to or less then 0 or the debt is less then or equal to 0 we can break out of the loop and check if the proper conditions are met to return true or false. This method runs in O(N)

Method Knapsack\_random(W,D,v,w):

This method is the random implementation of the knapsack problem and shares a lot of similarities with my greedy method. First we loop for a chosen number of times, for my method I decided on 10000, then I use the python zip function to zip the w and v list together then use the shuffle method from random to randomly shuffle the two list in the same order then I unzip the list and iterate from the beginning of the amount of items and subtract the weight and debt as we go if we run out of weight and debt then check if we came up with a solution if we did we just return true otherwise we keep iterating. The time complexity for this is interesting and is dependent on how many items we might have or how many times we randomly iterate so I would say it would be in this case O(N) N being how many times we iterate.

**Experimental results:**

 This lab was simple in creating the methods and I for the most part I didn’t have trouble. Here are some of the results from running all the methods

A picture containing window, sitting, black, table

Description automatically generatedA close up of text on a black background

Description automatically generatedA black sign with white text

Description automatically generatedThese are the solutions my algorithms got with the sets of problems given to us in the text documents. The most interesting to see was the backtracking we see that as our set goes up the time taken to complete increase exponentially with the last problem set taking so long to run I excluded it from the time as I have tried leaving it running for several hours and it not finishing. The Greedy algorithms were very fast, all of which taking less then time function could compute to complete but the drawback being it not always finding a solution, the random algorithm times are a bit random as well as its results, as I increased the amount of random iterations it go more and more accurate but taking longer although it is possible for all of the solutions to be found on the first try and be running time of O(1) although highly unlikely. The Random algorithm also didn’t find a solution every time it would vary and its possible for it to not find one even if there is one.

**Conclusions:**

This lab showed the usability and context of when and how to use different algorithmic designs, as well as the pros and cons of using them. First are backtracking is essential in finding a solution and will always do so if one exist but it can take a very long time sometimes so long it would be almost impossible to compute. The greedy algorithm fast and efficient does not always find a solution, but can be used as a good indicator if a solution exist or not. The random algorithm is very random, it has the possibility of finding a solution in O(1) or not finding one at all and as the sets get bigger its get unrealistic to use random, for example or last set of 50 items has 50! Different possibilities and to try every one randomly would take much more iterations then that and 50! Factorial is astronomically huge number that would be almost impossible to test all of them or if there’s a solution in those possibilities.

**Appendix:**

import numpy as np

from random import shuffle

import time

def knapsack\_backtracking(W,D,v,w):

if W < 0:

return False

if D < 0:

return True

if not v or not w:

return False

return knapsack\_backtracking(W-w[0],D-v[0],v[1:],w[1:]) or knapsack\_backtracking(W,D,v[1:],w[1:])

def knapsack\_greedy(W,D,v,w):

v = np.array(v)

w = np.array(w)

wvr = v/w

ind = np.argsort(-wvr)

for i in range(len(wvr)):

if W <= 0 or D <= 0:

break

W = W - w[ind[i]]

D = D - v[ind[i]]

if W >= 0 and D <= 0:

return True

return False

def knapsack\_random(W,D,v,w):

for i in range(10000):

wv = list(zip(w,v))

shuffle(wv)

w, v = zip(\*wv)

for j in range(len(w)):

if W <= 0 or D <= 0:

break

W = W - w[j]

D = D - v[j]

if W >= 0 and D <= 0:

return True

return False

if \_\_name\_\_ == "\_\_main\_\_":

print('\n \*\*\*\*\*\*\*\*\*\*\* Problem Set 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*')

W = 35

D = 221

w = [10, 6, 10, 6, 14, 8, 5, 13, 4, 1]

v = [39, 47, 47, 29, 71, 22, 50, 29, 51, 20]

start = time.time()

print("Can this problem be solved using random algorithim?",knapsack\_random(W,D,v,w))

end = time.time()

print("this algorithim took ", end-start, " seconds")

#print("Can this problem be solved using backtracking?",knapsack\_backtracking(W,D,v,w)

#print("Can this problem be solved using greedy algorithim?",knapsack\_greedy(W,D,v,w))

#print("Can this problem be solved using random algorithim?",knapsack\_random(W,D,v,w))

print('\n \*\*\*\*\*\*\*\*\*\*\* Problem set 2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*')

W = 10

D = 130

w = [10, 6, 10, 6, 14, 8, 5, 13, 4, 1]

v = [39, 47, 47, 29, 71, 22, 50, 29, 51, 20]

start = time.time()

print("Can this problem be solved using random algorithim?",knapsack\_random(W,D,v,w))

end = time.time()

print("this algorithim took ", end-start, " seconds")

#print("Can this problem be solved using backtracking?",knapsack\_backtracking(W,D,v,w)

#print("Can this problem be solved using greedy algorithim?",knapsack\_greedy(W,D,v,w))

#print("Can this problem be solved using random algorithim?",knapsack\_random(W,D,v,w))

print('\n \*\*\*\*\*\*\*\*\*\*\* Problem set 3 \*\*\*\*\*\*\*\*\*\*\*\*\*\*')

W = 102

D = 404

w = [10, 8, 5, 6, 9, 13, 13, 14, 13, 14, 6, 11, 12, 5, 13, 11, 9, 10, 14, 9]

v = [47, 15, 7, 17, 29, 12, 45, 24, 26, 10, 37, 38, 14, 35, 44, 37, 27,45, 36, 40]

start = time.time()

print("Can this problem be solved using random algorithim?",knapsack\_random(W,D,v,w))

end = time.time()

print("this algorithim took ", end-start, " seconds")

#print("Can this problem be solved using backtracking?",knapsack\_backtracking(W,D,v,w)

#print("Can this problem be solved using greedy algorithim?",knapsack\_greedy(W,D,v,w))

#print("Can this problem be solved using random algorithim?",knapsack\_random(W,D,v,w))

print('\n \*\*\*\*\*\*\*\*\*\*\* Problem set 4 \*\*\*\*\*\*\*\*\*\*\*\*\*\*')

W = 150

D = 600

w = [10, 14, 4, 5, 8, 12, 5, 7, 7, 11, 9, 5, 10, 14, 4, 4, 14, 7, 8, 9]

v = [39, 49, 47, 40, 20, 27, 31, 34, 17, 10, 29, 36, 41, 48, 45, 24, 15,17, 14, 40]

start = time.time()

print("Can this problem be solved using random algorithim?",knapsack\_random(W,D,v,w))

end = time.time()

print("this algorithim took ", end-start, " seconds")

#print("Can this problem be solved using backtracking?",knapsack\_backtracking(W,D,v,w)

#print("Can this problem be solved using greedy algorithim?",knapsack\_greedy(W,D,v,w))

#print("Can this problem be solved using random algorithim?",knapsack\_random(W,D,v,w))

print('\n \*\*\*\*\*\*\*\*\*\*\* Problem set 5 \*\*\*\*\*\*\*\*\*\*\*\*\*\*')

W = 200

D = 960

w = [ 8, 13, 13, 9, 5, 14, 13, 4, 8, 7, 13, 8, 12, 9, 13, 8, 5,

9, 5, 7, 4, 7, 13, 13, 6, 8, 4, 5, 9, 10, 5, 4, 6, 10,

7, 9, 13, 14, 12, 5, 10, 7, 9, 12, 9, 10, 5, 8, 11, 9]

v = [21, 26, 25, 23, 42, 32, 45, 33, 40, 20, 44, 13, 9, 31, 47, 21, 31,

18, 41, 36, 32, 43, 20, 40, 23, 16, 10, 44, 38, 6, 11, 13, 43, 7,

35, 21, 7, 25, 47, 34, 33, 46, 26, 17, 23, 28, 42, 16, 28, 30]

start = time.time()

print("Can this problem be solved using random algorithim?",knapsack\_random(W,D,v,w))

end = time.time()

print("this algorithim took ", end-start, " seconds")

#print("Can this problem be solved using backtracking?",knapsack\_backtracking(W,D,v,w))

#print("Can this problem be solved using greedy algorithim?",knapsack\_greedy(W,D,v,w))

#print("Can this problem be solved using random algorithim?",knapsack\_random(W,D,v,w))

**I, Estevan Ramos, certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.**