Summer 2019

Laboratory 6 – Algorithm Design Techniques

CS 2302 – Data structures summer 2019

Dilan Ramirez

INSTRUCTOR: OLAC FUENTES

TEACHING ASSISTANTS (TA): ISMAEL VILLANUEVA-MIRANDA

AUGUST 6, 2019

Contents

[Introduction 2](#_Toc16017666)

[Implementation 2](#_Toc16017667)

[Exercise 1 3](#_Toc16017668)

[Running Time 3](#_Toc16017669)

[Experimental results 3](#_Toc16017670)

[Appendix 4](#_Toc16017671)

[Exercise 2 4](#_Toc16017672)

[Running Time 4](#_Toc16017673)

[Experimental results 5](#_Toc16017674)

[Appendix 6](#_Toc16017675)

[Exercise 3 6](#_Toc16017676)

[Running Time 6](#_Toc16017677)

[Experimental results 7](#_Toc16017678)

[Appendix 8](#_Toc16017679)

[Exercise 3.2 8](#_Toc16017680)

[Running Time 8](#_Toc16017681)

[Experimental results 9](#_Toc16017682)

[Appendix 9](#_Toc16017683)

[Exercise 4 10](#_Toc16017684)

[Running Time 10](#_Toc16017685)

[Experimental results 11](#_Toc16017686)

[Appendix 11](#_Toc16017687)

[Conclusion 11](#_Toc16017688)

[Honesty Certification 12](#_Toc16017689)

# Introduction

Experiment with three algorithm design techniques applied to the solution of three version of the knapsack problem.

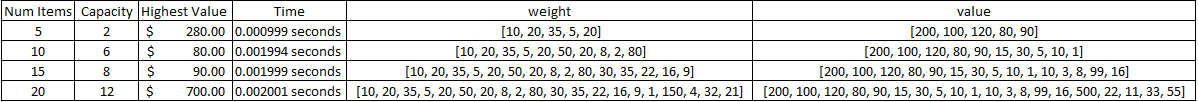
1. Implement a backtracking algorithm that solves the optimization 0-1 knapsack problem. Instead of deciding whether we can take items worth a predefined amount of money, as described in class, in this version of the problem you need to find the highest-value load that can fit in the knapsack.
2. Implement a greedy algorithm that solves the optimization continuous knapsack problem. This problem is identical the previous one, except that in this case we can take fractions of items. For example, if we take 3/4 of an item that has value 2 and weight 3, the value of the fraction would be 3/2 and its weight would be 9/4.\
3. Implement a randomized algorithm that solves the optimization 0-1 knapsack problem. You should generate many random permutations of the items and, for every permutation, add the items to the knapsack one by one until the capacity is reached. At the end, return the permutation that resulted in the largest value.
4. Implement a dynamic programming algorithm that solves the optimization integer knapsack problem. In this case, the thief can take multiple instances of an item. As before, you need to find the highestvalue load that can fit in the knapsack. Hint: This problem is similar to the minimum coin problem described in class.

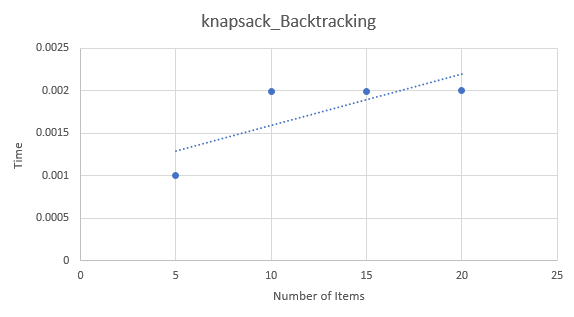
# Implementation

It uses two lists, one is for the weight of the items that the thief can steal, and the other is for the value of the items. First, it checks that the capacity of the knapsack is not zero and the length of the value’s list is not 0. Then, it checks the length of the two list. If the length is different, it returns 0 because there cannot be an item without value or weight. After that, it starts comparing the value of every item to get the highest value item that can be loaded in the knapsack. The last recursive call get the maximum item that was loaded in the bag.

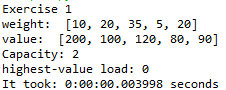
# Exercise 1

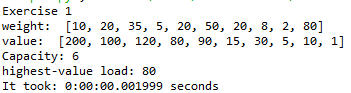
## Running Time

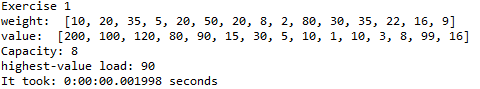


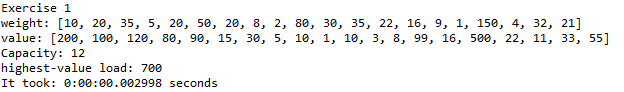


## Experimental results









## Appendix

def knapsack\_Backtracking(capacity,weight,value,n):

if capacity == 0 or n == 0:

return 0

if len(weight) != len(value):

return 0

if weight[n-1] > capacity:

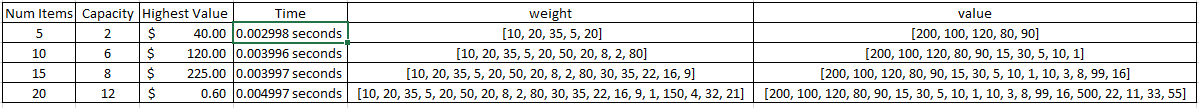
return knapsack\_Backtracking(capacity, weight,value,n-1)

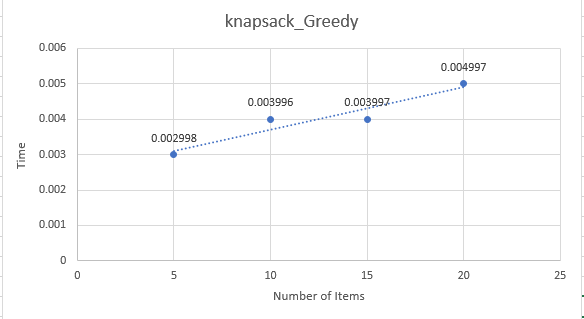
else:

return max(value[n-1] + knapsack\_Backtracking(capacity-weight[n-1], weight, value, n-1), knapsack\_Backtracking(capacity, weight, value,n-1))

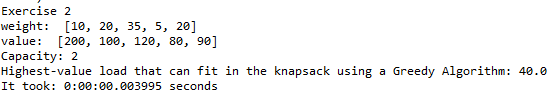
# Exercise 2

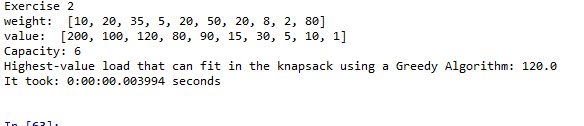
## Running Time

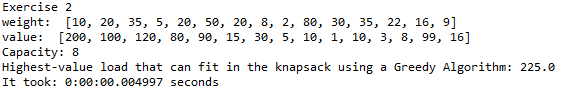


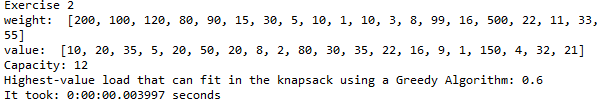


## Experimental results









## Appendix

def knapsack\_Greedy(capacity, weight, values):

items = []

for i in range(len(weight)):

items.append((weight[i], values[i], i))

sorted(items, reverse=True) # This list is sorted to make it easier

highest\_value = 0

for i in items:

curWeight = int(i[0])

curValue = int(i[1

if capacity - curWeight >= 0:

capacity -= curWeight

highest\_value += curValue

else:

fraction = capacity / curWeight

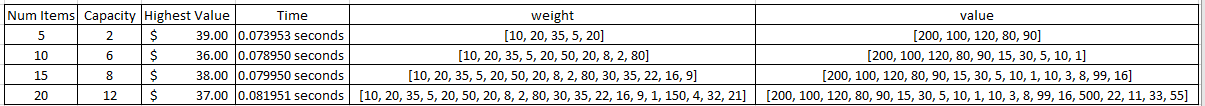
highest\_value += curValue \* fraction

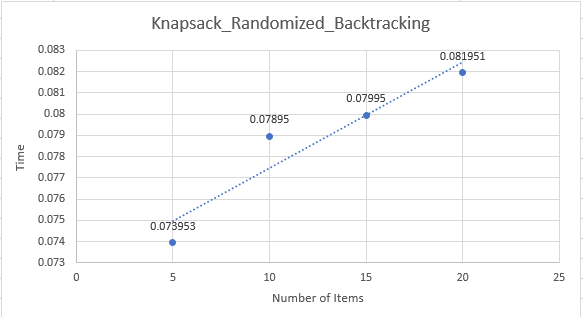
capacity = int(capacity-(curWeight \* fraction))

return highest\_value

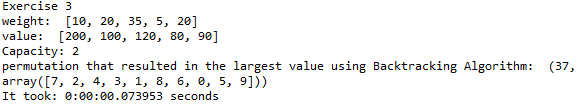
# Exercise 3

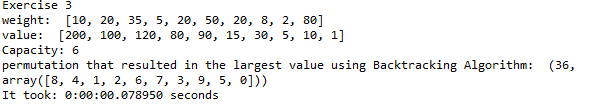
## Running Time

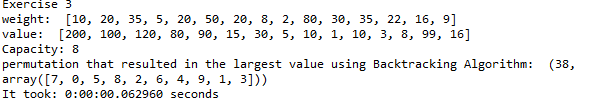


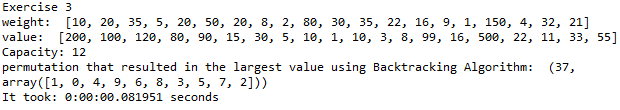


## Experimental results









## Appendix

def knapsack\_randomized\_Backtracking(capacity, numitems, tries=10):

highest\_value, pos = -2302, 0

perm, sack = [], []

for i in range(tries):

weight = np.random.permutation(numitems)

values = np.random.permutation(numitems)

perm.append(values) sack.append(knapsack\_Backtracking(capacity,weight,values,len(values)))

for i in range(len(sack)):

if sack[i] > highest\_value:

highest\_value, pos = sack[i], i

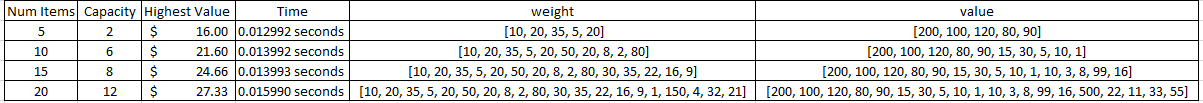
for j in range(len(perm)):

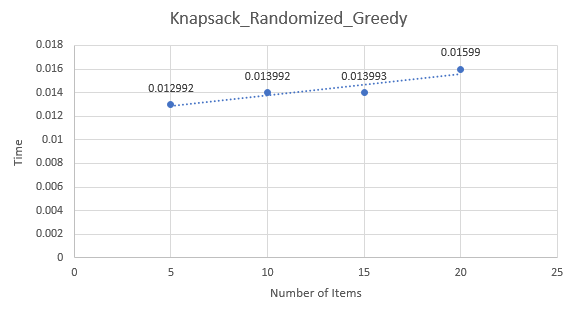
if j == pos:

return highest\_value, perm[j**]**

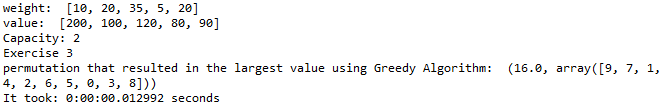
# Exercise 3.2

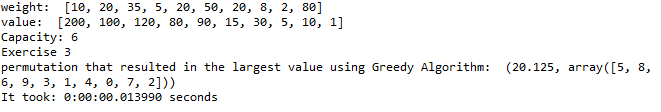
## Running Time

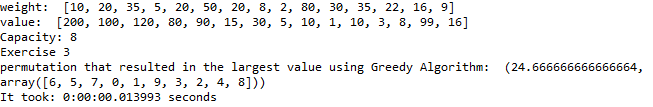


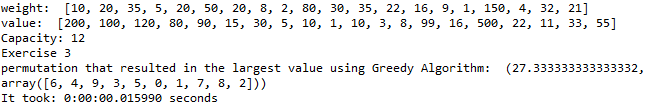


## Experimental results









## Appendix

def knapsack\_randomized\_Greedy(capacity, numitems, tries=10):

highest\_value, pos = -2302, 0

perm, sack = [], []

for i in range(tries):

weight = np.random.permutation(numitems)

values = np.random.permutation(numitems)

perm.append(values)

sack.append(knapsack\_Greedy (capacity,weight,values,len(values)))

for i in range(len(sack)):

if sack[i] > highest\_value:

highest\_value, pos = sack[i], i

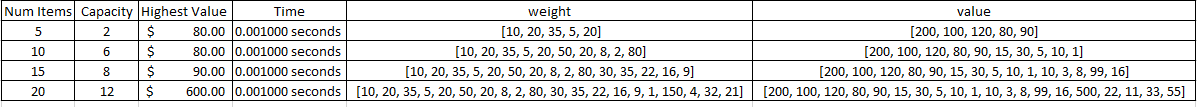
for j in range(len(perm)):

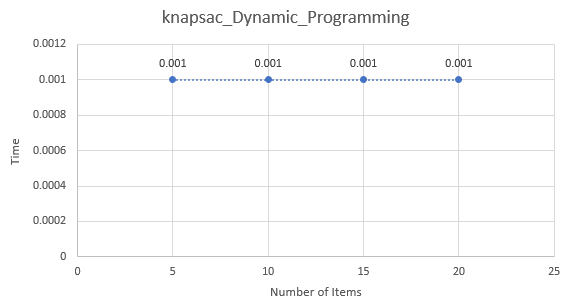
if j == pos:

return highest\_value, perm[j**]**

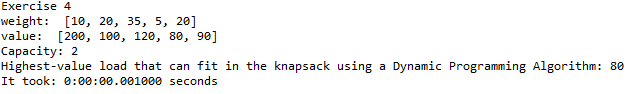
# Exercise 4

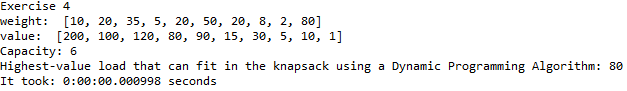
## Running Time

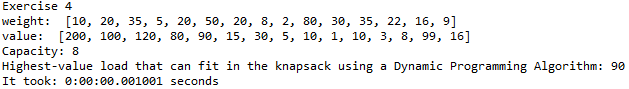


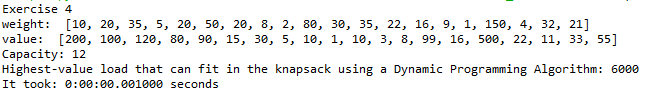


## Experimental results









## Appendix

def knapsac\_Dynamic\_Programming(numitems, weight, values, total):

knapsack = [0 for i in range(numitems + 1)

for i in range(numitems + 1):

for j in range(total):

if weight[j] <= i:

knapsack[i] = max(knapsack[i], knapsack[i-weight[j]] + values[j])

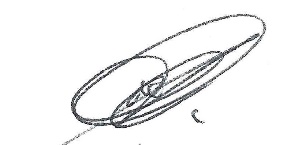
return knapsack[numitems]

# Conclusion

By doing this exercise, I learned how implement a backtracking, Greedy, Randomization, and Dynamic programming algorithms work in a real-life problem even though it is not the best real-life problem. I also learned how a backtracking algorithm works. I had some problems in this lab but overcome them. In conclusion, after comparing the running times of the algorithms for various parameter values I conclude that dynamic programming is the best one due to its running time which is constant.

# Honesty Certification

I 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 provide inappropriate assistance to any student in the class.

 08/ 06 / 2019

Dilan Ramirez Date