Nathan Aun

7/30/19

Lab #6 – Algorithm Design Techniques

CS2302 – Data Structures

Summer 2019

Introduction

The problem posed for this lab was the famous knapsack problem. In it, you are trying to help a thief make the most money from stealing items, but the thief can only carry so many. So you have to create an algorithm that will help the thief make the most money given the weight limit on their knapsack. This lab requires us to solve 4 slightly different variations on the problem; these include the 1-0 knapsack problem, where you can either take or not take an item, the fractional knapsack problem, where you can take parts of items, and lastly the 1-0 knapsack problem where you are allowed to take multiple of the same item.

Proposed Solution

The algorithm for the initial 1-0 knapsack problem will be a backtracking algorithm that will generate all possible outcomes of the problem and then return the one that has the highest value while staying within the constraints of the problem. This is a very slow solution as it has a runtime of O(2^n) but it one of the most intuitive solutions to this problem.

The solution to the fractional knapsack problem utilises sorting in order to get the best possible values. We find the ratio of value to weight and sort by the highest (that is, the one that has the most value for the least weight) and from there it adds weights until the weight condition is met. For the last value, it is likely that we will need to take a part of it instead of the full thing so the program will calculate what percentage we’re taking and adds that percentage of the value to the answer.

The second solution to the initial 1-0 knapsack problem utilises randomization in order to solve the problem. It will generate a default of 10,000 random permutations of all items in the room and will go through and add items until the conditions are met. If the new total is better than the previous one, then it will save it as the max and when it’s done all permutations it will end. This one will not necessarily always give the best solution but depending on how many items there are it has a decent chance to get close.

The final thing to be implemented is a solution using dynamic programming to solve the problem when you can take multiple of the same item. This goes through and adds items to an array based on weight and if the new value is better than the previous one it will add it, if not it will keep the old value. This solution is probably the best one out of the lot in terms of speed, however it does only solve a specific knapsack problem and not the main one that we usually talk about.

I will test these using very large sets of “items” generated by a method called generateItems. It will make a user defined amount of items with weights and values that are randomly assigned, which will allow us to generalize pretty easily based on our results.

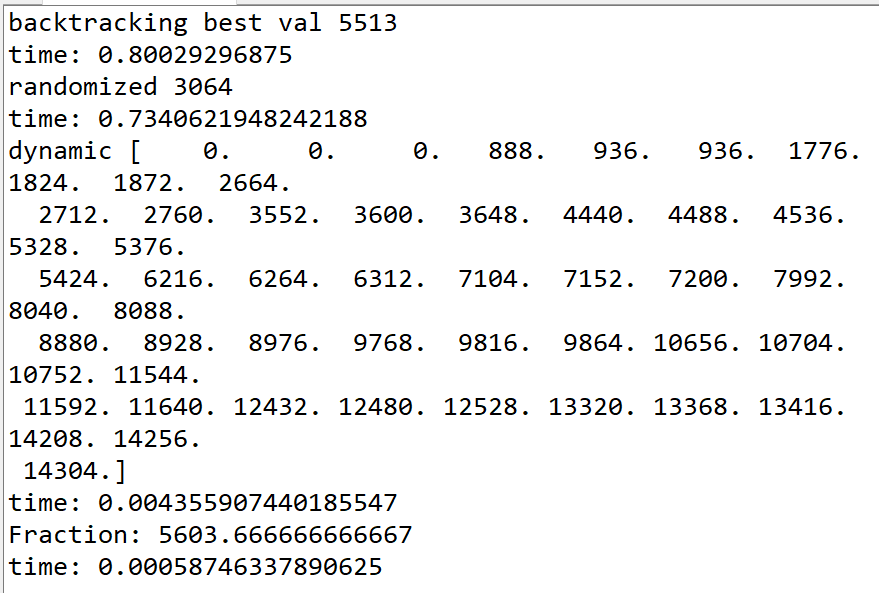
Conclusions

Overall, the fractional solution tended to be the overall best solution to the problem, with the fastest runtime and the second highest value achieved out of any of the other algorithms. This algorithm is particularly unrealistic however, because while it might make sense to take fractions of certain items (like a pizza or cake) it would not make sense at all to take part of a TV or part of a guitar or something else that is similarly valuable together and useless when cut into parts. So while it did get the best results, it is not very reasonable to assume that this is the best possible version of this problem.

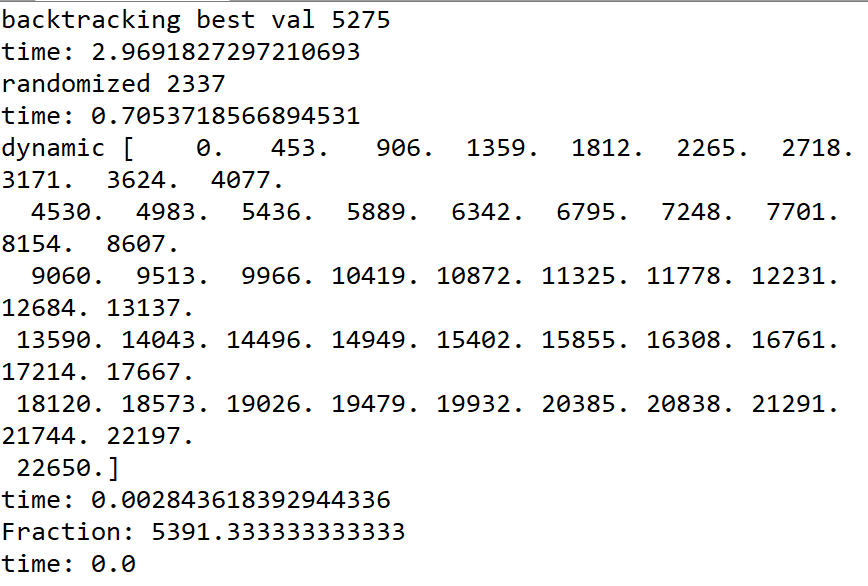
The next best solution was the dynamic programming one, and this one had the second best runtime out of all the algorithms and it had the highest value achieved by any of the algorithms. The problem is, that the problem that it solves is also a bit unrealistic, depending on the context of the robbery. If the robber is robbing from a store, then this approach makes sense since there would be multiple of each item, but where it doesn’t make sense is robbing a house or somewhere that isn’t likely going to have multiple of each item.

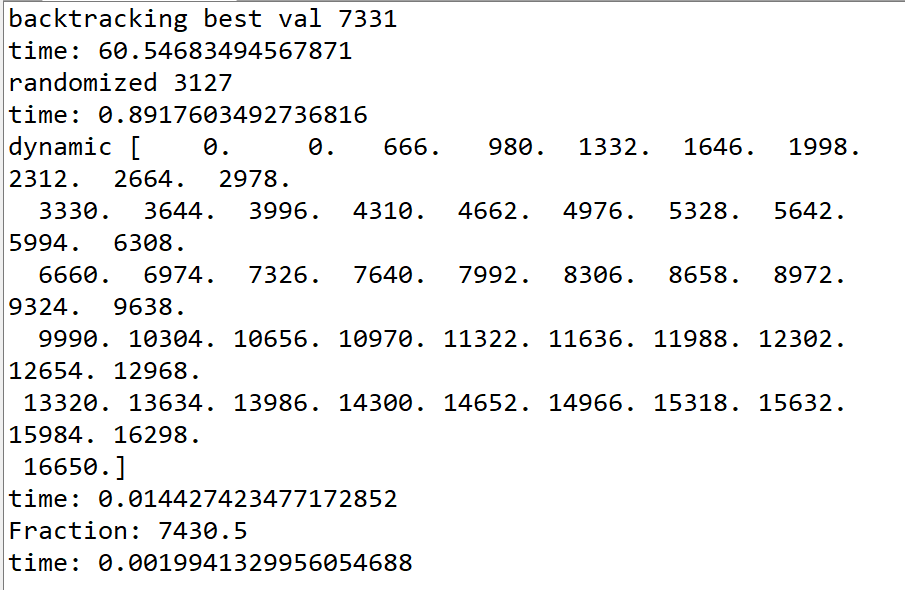
The next best in terms of time is the randomized solution, however it usually did not get the right answer and in some cases it was quite a bit off. Still it was fast and it does solve the 1-0 knapsack problem, it just won’t necessarily give the optimal solution.

Lastly the worst in terms of time was the backtracking solution, which makes sense because it has to go through every single possible outcome in order to determine which one is the best. It makes sense and it does accurately solve the 1-0 knapsack problem well, but it just takes a very long time to do so.



Output with 100 items

Output with 150 items



Output with 200 items

Appendix

Source code is as follows:

'''

report link: https://docs.google.com/document/d/1LBB4Qh8p-fYWYVFu27cLm6R1-Zt8olp0eli6X7GZhhA/edit?usp=sharing

Course: CS2302 Data Structures

Author: Nathan Aun

Assignment: Lab 6 - Algorithm Design Techniques

Instructor: Dr. Fuentes

TA: Ismael Villanueva-Miranda

Date of Last Modification:  8/4/2019

Purpose of Program: To solve variations of the knapsack problem using a variety of different techniques

'''

import math

import numpy as np

import random

import time

#recieves a list of items in [[weight, value],[weight2,value2]] format, a target weight and 0

# and returns the highest possible value from taking items (only taking 1)

def backtracking(items, remainingWeight, currentVal):

    if remainingWeight < 0:

        return -math.inf

    if remainingWeight == 0 or items == []:

        return currentVal

    j = max(backtracking(items[1:], remainingWeight-items[0][0], currentVal+items[0][1]), backtracking(items[1:], remainingWeight, currentVal))

    return j

#Same problem as the last one, except we can take fractions of an item. It finds all the value-weight ratios of the items

#array and sorts via the ratio, then we add them to the knapsack by best ratio until we cannot add a full one. When we can't

#we add whatever fraction of it we can

def fraction(items, weight):

    for i in items:

        ratio = i[1] / i[0]

        i.append(ratio)

    items.sort(key = lambda x: x[2])

    val = 0

    while weight > 0 and items != []:

        if weight - items[-1][0] < 0:

            ratio = weight/items[-1][0]

            val += items[-1][1] \* ratio

            weight = 0

        else:

            weight -= items[-1][0]

            val += items[-1][1]

            items.pop(-1)

    return val

#Randomly selects a default of 10,000 permutations and finds the best one and the value obtained from it

def randomized(items, weight, amountOfRuns = 10000):

    temp = items

    currVal = -math.inf

    goodPermutation = items

    for i in range(amountOfRuns):

        currWeight = weight

        val = 0

        temp = np.random.permutation(temp)

        #print(temp)

        i = 0

        while currWeight >= 0 and i < len(temp):

            if currWeight - temp[i][0] < 0:

                if currVal < val:

                    currVal = val

                    #print(goodPermutation)

                    goodPermutation = temp

                break

            if currWeight - temp[i][0] == 0:

                val += temp[i][1]

                if currVal < val:

                    currVal = val

                    goodPermutation = temp

                break

            val += temp[i][1]

            currWeight -= temp[i][0]

            i+= 1

    return currVal

#uses dynamic programming to find the best value obtained for every possible weight from 0->weight

def dynamic(items, weight):

    arr = np.zeros(weight+1)

    items.sort(key = lambda x: x[0])

    for i in range(len(items)):

        for j in range(items[i][0],len(arr)):

            arr[j] = max(items[i][1] + arr[j-items[i][0]], arr[j])

        #print(arr)

    return arr

#generates a list of random items of a given size

def generateItems(amount = 100):

  L = []

  temp = []

  for i in range(amount):

    temp = [random.randint(1,100), random.randint(1,1000)]

    L.append(temp)

  return L

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

    #items = [[6,25],[8,42],[14,65],[18,95],[20,100]]

    #items = [[10,60],[20,100],[30,120]]

    items = generateItems(100)

    print(items)

    maxWeight = 50

    before = time.time()

    print("backtracking best val", backtracking(items,maxWeight,0))

    print("time:", time.time()- before)

    before = time.time()

    #items = [[6,25],[8,42],[14,65],[18,95],[20,100]]

    #items = [[10,60],[20,100],[30,120]]

    print("randomized",randomized(items,maxWeight))

    print("time:", time.time()-before)

    before = time.time()

    print("dynamic",dynamic(items,maxWeight))

    print("time:", time.time() - before)

    before = time.time()

    print("Fraction:",fraction(items,maxWeight))

    print("time:", time.time() - before)

Statement of Academic Honesty

“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 provided inappropriate assistance to any student in the class.”

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_