Assignment 4

BMI 2005

Snoop DAG

By: **Ryan Neff**
ryan.neff@icahn.mssm.edu

Due: 4/10/2019

Problem 1

In class we discussed the dynamic programming approach to the change problem, DPChange. The pseudocode is provided below. (Note that *M* is an integer (cents), **c** is a vector of denomination values (integer cents) in any particular order, and *d* is a positive integer corresponding to the number of denominations.)

```
DPChange(M, c, d):
    bestNumCoins[0] = 0
    for m = 1 to M
        bestNumCoins[m] = inf
        for i = 1 to d
            if m >= c[i]
                if bestNumCoins[m-c[i]] + 1 < bestNumCoins[m]
                    bestNumCoins[m] = bestNumCoins[m-c[i]] + 1
    return bestNumCoins[M]
```

Problem 1a

Convert this pseudocode to Python.

In [221]:
```
from collections import defaultdict

class InputError(Exception):
    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

def print_arr(arr):
    return "|"+".".join([str(a) for a in arr])+"|"

def DPChange(M,c,d=False,verbose=False):
    '''DPChange(M,c,d)
        inputs:
            M = an **INTEGER** corresponding to the number of cents to make change for
            c = a vector of **INTEGERS** of denomination values, in cents, in no particular order
            d = a positive integer corresponding to the number of denominations in c
        returns:
            bestNumCoins[M]
                The number of coins from vector c to make denomination M
    '''
    if type(M)!=int: raise InputError("DPChange()","M must be of type int, but got "+str(type(M)))
    if set([type(a) for a in c])!={int}: raise InputError("DPChange()","c must all be of type int")
    if d==False: d = len(c)
    if d and (d != len(c)): raise InputError("DPChange()","d must be equal to length c")
    if type(d)!=int: raise InputError("DPChange()","d must be of type int, but got "+str(type(d)))
    bestNumCoins = [0]
    tick = 0
    for m in range(1,M+1):
        bestNumCoins.append(float('inf'))
        for i in range(d):
            tick += 1
            if m >= c[i]:
                if bestNumCoins[m-c[i]] + 1 < bestNumCoins[m]:
                    bestNumCoins[m] = bestNumCoins[m-c[i]] + 1
                    if verbose>=2: print("array:",print_arr(bestNumCoins))
    if verbose>=1: print("runtime: " + str(tick))
    return bestNumCoins[M]
```

In [222]:
```
DPChange(13,[9,4,1])
```

Out[222]: 2

Problem 1b

What is the runtime of this algorithm?

In [223]:
```
DPChange(13,[4,9,1],d=3,verbose=1)
```

```
runtime: 39
```

Out[223]: 2

Answer: The runtime of DPChange is M * d, regardless of the number of coins returned as the best solution. This is from the double loop over 1->M and 1->d, and is shown in the output from tick in 1a.

Problem 1c

Modify your function to return not only the best number of coins but also the corresponding combination of denominations as a vector. Do this by create a new function DPChange_bt that is largely the same as DPChange but declares an array coinsUsed that keeps track of the coin on the best 'path' to a given entry m (between 1 and M+1). DPChange_bt should return both bestNumCoins and coinsUsed.

In [224]:
```
import copy

def DPChange_bt(M,c,d=False,verbose=False):
    '''DPChange(M,c,d)
        inputs:
            M = an **INTEGER** corresponding to the number of cents to make change for
            c = a vector of **INTEGERS** of denomination values, in cents, in no particular order
            d = a positive integer corresponding to the number of denominations in c
        returns:
            bestNumCoins[M]
                The number of coins from vector c to make denomination M
    '''
    if type(M)!=int: raise InputError("DPChange()","M must be of type int, but got "+str(type(M)))
    if set([type(a) for a in c])!={int}: raise InputError("DPChange()","c must all be of type int")
    if d==False: d = len(c)
    if d and (d != len(c)): raise InputError("DPChange()","d must be equal to length c")
    if type(d)!=int: raise InputError("DPChange()","d must be of type int, but got "+str(type(d)))
    bestNumCoins = [0]
    coinsUsed = [[0]*(d)]
    tick = 0
    for m in range(1,M+1):
        bestNumCoins.append(float('inf'))
        coinsUsed.append([0]*(d))
        for i in range(d):
            tick += 1
            if m >= c[i]:
                if bestNumCoins[m-c[i]] + 1 < bestNumCoins[m]:
                    bestNumCoins[m] = bestNumCoins[m-c[i]] + 1
                    coinsUsed[m] = copy.copy(coinsUsed[m-c[i]])
                    coinsUsed[m][i] += 1
                    if verbose>=3: print("coins:",coinsUsed[m],end=" ")
                    if verbose>=2: print("array:",print_arr(bestNumCoins))
    if verbose>=1: print("runtime: " + str(tick))
    return bestNumCoins[M], coinsUsed[M]
```

In [225]:
```
DPChange_bt(13,[9,4,1],d=3,verbose=1)
```

```
runtime: 39
```

Out[225]: (2, [1, 1, 0])

Problem 1d

Create a function printCoins that prints the sequence of coins used to get to M.

In [226]:
```
def printCoins(M,c,d=False):
    return(DPChange_bt(M,c,d,verbose=3))
```

```
In [227]: printCoins(13,[9,4,1],d=3)
```

```
coins: [0, 0, 1] array: |0.1|
coins: [0, 0, 2] array: |0.1.2|
coins: [0, 0, 3] array: |0.1.2.3|
coins: [0, 1, 0] array: |0.1.2.3.1|
coins: [0, 1, 1] array: |0.1.2.3.1.2|
coins: [0, 1, 2] array: |0.1.2.3.1.2.3|
coins: [0, 1, 3] array: |0.1.2.3.1.2.3.4|
coins: [0, 2, 0] array: |0.1.2.3.1.2.3.4.2|
coins: [1, 0, 0] array: |0.1.2.3.1.2.3.4.2.1|
coins: [1, 0, 1] array: |0.1.2.3.1.2.3.4.2.1.2|
coins: [1, 0, 2] array: |0.1.2.3.1.2.3.4.2.1.2.3|
coins: [1, 0, 3] array: |0.1.2.3.1.2.3.4.2.1.2.3.4|
coins: [0, 3, 0] array: |0.1.2.3.1.2.3.4.2.1.2.3.3|
coins: [1, 1, 0] array: |0.1.2.3.1.2.3.4.2.1.2.3.3.2|
runtime: 39
```

```
Out[227]: (2, [1, 1, 0])
```

Problem 1e

Has the runtime changed with addition of this backtrace? Why or why not?

Answer: We have now added a few more function calls that will add some time to the algorithm at some fractional rate of M*d dependent on the number of branches on the tree but have not changed the double loop. So we could say we now have an O(1.5*M*d) runtime. As shown by the runtime counter and the structure of the double loop, it is exactly the same M * d number of loop iterations as it was previously. The dynamic algorithm already substitutes in larger coins for smaller coins (the m - c[i] index accomplishes this) for each amount of currency as it works up from 1 cent to M so the loop doesn't change - but the time per optimal solution found does increase.

Take a look at the following runtime comparison between the algorithms with and without the trace. We can snoop in on the actual running time with the `%timeit` line magic in Jupyter.

```
In [228]: %timeit DPChange(420000,[98,108,97,122,101,32,105,116])
          %timeit DPChange_bt(420000,[98,108,97,122,101,32,105,116])
```

```
1 loop, best of 3: 1.1 s per loop
1 loop, best of 3: 1.62 s per loop
```

Problem 2

The Knapsack Problem is a famous computer science problem that is as follows: imagine you are carrying a knapsack with capacity to hold a total of weight W. You are selecting among n items with values {a_1, a_2, ... , a_n} with associated weights {w_1, w_2, ... , w*n}. Here the weights and values are all positive (but not necessarily unique). You wish to maximize the total value of the items you select (call this set A) not exceeding the given weight capacity, i.e. maximize sum*{a in A} such that sum w < W.

Problem 2a

We can reformulate this as a 2D dynamic programming problem as follows. Define T*{i,j} as the highest possible value sum considering items 1 through i and total weight capacity j (j <= W). What is the base case i.e. T*{0,j} for all j and T_{i,0} for all i?

Let:
T{i,j} = max(sum{a in A} such that sum{w} < W) where a=i and w=j

What is the base case?
T{0,j} for all j?
T{i,0} for all i?

Answer:

T{0,j} = 0. There are no items in the knapsack.
T{i,0} = 0. The knapsack cannot hold any items, regardless of their value, because the max weight is 0.

Problem 2b

What is the recursive statement?

In [229]:
```
#SOURCES (inspiration): https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

def recursive_knapsack(values, weights, total_weight,step=None):

    if step==None: #if we are the parent recursion
        step = len(weights) # set steps as maximum
        wv = sorted(zip(values,weights),key=lambda x:x[1]) #ensure the weights are sorted ascending
        weights = [w for v,w in wv] #get sorted weights
        values = [v for v,w in wv] #get values for each sorted weight

    ### BEGIN RECURSIVE STATEMENT ###

    if step==0 or total_weight==0: #base case. everything is 0
        return 0

    elif weights[step-1] > total_weight: #weight exceeds total, then we can't include it.
        # move directly to next step (no branching)
        return recursive_knapsack(values, weights,total_weight,step-1)

    else: #branch on whether to include the new item or not
        value_with_new_item = values[step-1] + recursive_knapsack(values,weights,total_weight-weights[step-1],step-1)
        value_without_new_item = recursive_knapsack(values,weights,total_weight,step-1)
        return max(value_with_new_item,value_without_new_item)

    ### END RECURSIVE STATEMENT ###
```

Problem 2c

Add an array to your function to hold the backtrace (e.g. some data structure that lets you keep track of the path to your optimal solution). What is the optimal set of items? An auxiliary printing function is one option to show this.

In [230]:
```
# SOURCES:
#    Inspired by https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/
#    Edited by me to implement best path tracking

import copy
from collections import defaultdict

def print_arr(arr):
    return "|"+".".join([str(a) for a in arr])+"|"

class HashedDict(dict):
    def __init__(self,index,input_dict):
        self.id = index
        self.dict = input_dict
    def __hash__(self):
        return self.id
    def __repr__(self):
        return str({self.id:self.dict})

def knapsack(values,weights,total_weight,verbose=0):
    '''knapsack(values,weights,total_weight)'''
    knap_arr = []
    items = []
    for v in range(len(values)+1):
        knap_arr.append([0]*(total_weight+2))
        items.append([{"bag":set()} for i in range(total_weight+2)])
        for w in range(total_weight+2):
            if v==0 or w==0: continue
            if weights[v-1] < w:
                if values[v-1] + knap_arr[v-1][w-weights[v-1]] > knap_arr[v-1][w]:
                    knap_arr[v][w] = values[v-1] + knap_arr[v-1][w-weights[v-1]]
                    items[v][w] = copy.deepcopy(items[v-1][w-weights[v-1]])
                    items[v][w]["bag"].add(HashedDict(v,{"val":values[v-1],"wt":weights[v-1]}))
                else:
                    knap_arr[v][w] = knap_arr[v-1][w]
                    items[v][w] = items[v-1][w]
            else:
                knap_arr[v][w] = knap_arr[v-1][w]
                items[v][w] = items[v-1][w]
    if verbose>=1:
        for line in range(1,len(knap_arr)):
            print("items="+str(line)+": " + print_arr(knap_arr[line]))
            if verbose>=2:
                print("\t\tknapsack:")
                for w in range(len(items[line])):
                    print("\t\t\t"+str(items[line][w]))
    return knap_arr[len(values)][total_weight+1], items[len(values)][total_weight+1]
```

In [231]:
```
values = [3, 5, 7, 10]
weights = [1, 2, 3, 4]
total_weight = 7

print("recursive knapsack solution: ",end="")
print(recursive_knapsack(values,weights,total_weight))
print("\n=======\n")
print("dynamic knapsack solution:\n")
print("ANSWER:",knapsack(values,weights,total_weight,verbose=2))
```

```
recursive knapsack solution: 18

=======

dynamic knapsack solution:

items=1: |0.0.3.3.3.3.3.3.3|
                knapsack:
                        {'bag': set()}
                        {'bag': set()}
                        {'bag': {{1: {'val': 3, 'wt': 1}}}}
                        {'bag': {{1: {'val': 3, 'wt': 1}}}}
                        {'bag': {{1: {'val': 3, 'wt': 1}}}}
                        {'bag': {{1: {'val': 3, 'wt': 1}}}}
                        {'bag': {{1: {'val': 3, 'wt': 1}}}}
                        {'bag': {{1: {'val': 3, 'wt': 1}}}}
items=2: |0.0.3.5.8.8.8.8.8|
                knapsack:
                        {'bag': set()}
                        {'bag': set()}
                        {'bag': {{1: {'val': 3, 'wt': 1}}}}
                        {'bag': {{2: {'val': 5, 'wt': 2}}}}
                        {'bag': {{1: {'val': 3, 'wt': 1}}, {2: {'val': 5, 'wt': 2}}}}
                        {'bag': {{1: {'val': 3, 'wt': 1}}, {2: {'val': 5, 'wt': 2}}}}
                        {'bag': {{1: {'val': 3, 'wt': 1}}, {2: {'val': 5, 'wt': 2}}}}
                        {'bag': {{1: {'val': 3, 'wt': 1}}, {2: {'val': 5, 'wt': 2}}}}
                        {'bag': {{1: {'val': 3, 'wt': 1}}, {2: {'val': 5, 'wt': 2}}}}
items=3: |0.0.3.5.8.10.12.15.15|
                knapsack:
                        {'bag': set()}
                        {'bag': set()}
                        {'bag': {{1: {'val': 3, 'wt': 1}}}}
                        {'bag': {{2: {'val': 5, 'wt': 2}}}}
                        {'bag': {{1: {'val': 3, 'wt': 1}}, {2: {'val': 5, 'wt': 2}}}}
                        {'bag': {{1: {'val': 3, 'wt': 1}}, {3: {'val': 7, 'wt': 3}}}}
                        {'bag': {{2: {'val': 5, 'wt': 2}}, {3: {'val': 7, 'wt': 3}}}}
                        {'bag': {{1: {'val': 3, 'wt': 1}}, {2: {'val': 5, 'wt': 2}}, {3: {'val': 7, 'wt': 3}}}}
                        {'bag': {{1: {'val': 3, 'wt': 1}}, {2: {'val': 5, 'wt': 2}}, {3: {'val': 7, 'wt': 3}}}}
items=4: |0.0.3.5.8.10.13.15.18|
                knapsack:
                        {'bag': set()}
                        {'bag': set()}
                        {'bag': {{1: {'val': 3, 'wt': 1}}}}
                        {'bag': {{2: {'val': 5, 'wt': 2}}}}
                        {'bag': {{1: {'val': 3, 'wt': 1}}, {2: {'val': 5, 'wt': 2}}}}
                        {'bag': {{1: {'val': 3, 'wt': 1}}, {3: {'val': 7, 'wt': 3}}}}
                        {'bag': {{1: {'val': 3, 'wt': 1}}, {4: {'val': 10, 'wt': 4}}}}
                        {'bag': {{1: {'val': 3, 'wt': 1}}, {2: {'val': 5, 'wt': 2}}, {3: {'val': 7, 'wt': 3}}}}
                        {'bag': {{1: {'val': 3, 'wt': 1}}, {2: {'val': 5, 'wt': 2}}, {4: {'val': 10, 'wt': 4}}}}
ANSWER: (18, {'bag': {{1: {'val': 3, 'wt': 1}}, {2: {'val': 5, 'wt': 2}}, {4: {'val': 10, 'wt': 4}}}})
```

Problem 3

Here we have an incomplete class definition of a graph along with the following incomplete methods:

- addEdge which connects a node v to a pre-existing node u
- toplogicalSortUtil which is a helper function for topologicalSort that prints out the topological sort of the graph.

FYI: defaultdict is a class from collections that enables you to create and append to the list corresponding to a key that may not exist in that dictionary yet in a single command. There are four areas in the class that you need to fill out (labeled (1), (2), (3) and (4) in the comments). Note that this will be helpful in solving the Rosalind lab problems due 4/10.

In [232]:
```
#Python program to print topological sorting of a DAG
from collections import defaultdict

#Class to represent a graph
class Graph:
    def __init__(self,vertices):
        self.graph = defaultdict(list) # defaultdict of adjacency List
        # (1) create an attribute V that returns the number of
        # vertices in the graph according to the init parameter
        # vertices
        self.V = vertices #this is the number of nodes in the graph

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A recursive function used by topologicalSort
    def topologicalSortUtil(self,v,visited,stack):

        # Mark the current node as visited.
        visited[v] = True

        # Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
            if visited[i] == False:  #if not yet visited
                self.topologicalSortUtil(i,visited,stack)  #mark as visited
                # Push current vertex to stack which stores result

        # (2) one of the following two commands will give the
        # correct topological sorting, the other reverses the order;
        # which is which and why?
        # (a) stack.insert(len(stack),v)
        # (b) stack.insert(0,v)
        stack.insert(0,v)
            #ANSWER: because we are traversing from the first node visited to the last (top-down)
                    # if we started at the bottom of the stack and built upward, we would miss other neighbors
                    # in separate branches than our own

    # The function to do Topological Sort. It uses recursive
    # topologicalSortUtil()
    def topologicalSort(self):
        # Mark all the vertices as not visited
        # (3) what should visited be defined a to create a list
        # of length vertices of False values, indicating no nodes
        # have yet been reached?
        visited = [False] * (self.V)
        stack = []

        # Call the recursive helper function to store Topological
        # Sort starting from all vertices one by one

        # (4) what should the argument to the range function be
        # to ensure you iterate over all vertices in the graph?
        for i in range(self.V):
            if visited[i] == False:
                self.topologicalSortUtil(i,visited,stack)

        # Print contents of the stack
        print(stack)

# example call
g = Graph(6)
g.addEdge(5, 2);
g.addEdge(5, 0);
g.addEdge(4, 0);
g.addEdge(4, 1);
g.addEdge(2, 3);
g.addEdge(3, 1);

print("Following is a Topological Sort of the given graph")
g.topologicalSort()
```

```
Following is a Topological Sort of the given graph
[5, 4, 2, 3, 1, 0]
```