def maxVal(toConsider, avail):
    """Assumes toConsider a list of items, avail a weight
    Returns a tuple of the total weight of a solution to the
    0/1 knapsack problem and the items of that solution"
    if toConsider == [] or avail == 0:
        result = (0, ()
    elif toConsider[0].getWeight() > avail:
        #Explore right branch only
        result = maxVal(toConsider[1:], avail)
    else:
        nextItem = toConsider[0]
        #Explore left branch
        withVal, withToTake = maxVal(toConsider[1:],
                                    avail - nextItem.getWeight())
        withVal += nextItem.getValue()
        #Explore right branch
        withoutVal, withoutToTake = maxVal(toConsider[1:],
                                            avail)
        #Choose better branch
        if withVal > withoutVal:
            result = (withVal, withToTake + (nextItem,))
        else:
            result = (withoutVal, withoutToTake)
    return result

def bigTest(numItems, maxValue, maxWeight):
    items = buildManyItems(numItems, 10, 10)
    val, taken = maxVal(items, 40)
    print 'Items Taken'
    for item in taken:
        print item
    print 'Total value of items taken =', val

    callsMade = []
    numberOfItems = [5,10,15,20,25]
    for numItems in numberOfItems:
        numCalls = 0
        bigTest(numItems, 10, 10)
        callsMade.append(numCalls)
    pylab.plot(numberOfItems, callsMade)
    pylab.xlabel('Number of Items')
    pylab.ylabel('Number of Calls')
    pylab.semilogy()
def printPath(path):
    # a path is a list of nodes
    result = ''
    for i in range(len(path)):
        if i == len(path) - 1:
            result = result + str(path[i])
        else:
            result = result + str(path[i]) + '->
    return result

def DFS(graph, start, end, path = [], shortest = None):
    # assumes graph is a Digraph
    # assumes start and end are nodes in graph
    path = path + [start]
    print 'Current dfs path:', printPath(path)
    if start == end:
        return path
    for node in graph.childrenOf(start):
        if node not in path: # avoid cycles
            if shortest == None or len(path)<len(shortest):
                newPath = DFS(graph,node,end,path,shortest)
                if newPath != None:
                    shortest = newPath
    return shortest

def BFS(graph, start, end, pathQueue = []):
    # assumes graph is a Digraph
    # assumes start and end are nodes in graph
    initPath = [start]
    pathQueue.append(initPath)
    visited = [start]
    while len(pathQueue) != 0:
        # Get and remove oldest element in pathQueue
        tmpPath = pathQueue.pop(0)
        lastNode = tmpPath[len(tmpPath) - 1]
        print 'Current BFS path:', printPath(tmpPath)
        if lastNode == end:
            return tmpPath
        for linkNode in graph.childrenOf(lastNode):
            if linkNode not in visited:
                newPath = tmpPath + [linkNode]
                pathQueue.append(newPath)
    return None