HW #5 Solution

Problem: 0-1 Knapsack Problem

1. Initialize the problem with the given weights and values.
2. For each item, calculate the profit-to-weight ratio.
3. Sort the items based on the profit-to-weight ratio in descending order.
4. Create a table to keep track of the maximum profit at each weight.
5. For each weight, check if the current item can be included.
6. If the current weight is less than or equal to the weight of the current item, include it and calculate the new profit.
7. If the current weight is greater than the weight of the current item, do not include it.
8. Update the maximum profit at the current weight.
9. Repeat steps 4-8 until all items are considered.
10. The maximum profit is the value in the last row of the table.

Example:

- Item 1: Weight = 2, Value = 5
- Item 2: Weight = 3, Value = 7
- Item 3: Weight = 4, Value = 9
- Item 4: Weight = 5, Value = 11
- Weights: 0, 1, 2, 3, 4, 5
- Maximum Profit:
  - Weight 0: 0
  - Weight 1: 5
  - Weight 2: 7
  - Weight 3: 9
  - Weight 4: 11
  - Weight 5: 11

Conclusion: The maximum profit is 11.
Chapter 8
Exercise 33. Let S and T be two arrays of n numbers that are already in nondecreasing order. Write an algorithm that finds the median of all 2n numbers whose time complexity is in $\Theta(\lg n)$.

To get $\Theta(\lg n)$, we need to be able to discard half of the remaining items each iteration.

<table>
<thead>
<tr>
<th>S_left</th>
<th>S_mid</th>
<th>S_right</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T_left</th>
<th>T_mid</th>
<th>T_right</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

General Idea is to recursively compare the middle item of the remaining part of S (between S_left and S_right) with the middle item of the remaining part of T (between T_left and T_right).

```java
num recursiveMedian(S, S_left, S_right, T, T_left, T_right) {
    If S[S_mid] == T[T_mid] then
        return S[S_mid]
    else if S[S_mid] < T[T_mid] then
        return recursiveMedian(S, S_mid, S_right, T, T_left, T_mid)
    else
        return recursiveMedian(S, S_left, S_mid, T, T_mid, T_right)
}
```

The initial call would be recursiveMedian(S, 1, n, T, 1, n)

Exercise 36.
Interpolation Search - To be effective, the items must be relatively equally spaced.

Binary Search Tree - useful if the items are to be added dynamically and need to access some or all of them in sorted order.

Hashing - useful if you need to find items quickly, but never need to iterate through all items.
class Node(object):
    """ Node objects hold state information about for each node in the 0-1 Knapsack problem""

    def __init__(self, level, profit, weight, bound):
        self.level = level
        self.profit = profit
        self.weight = weight
        self.bound = bound
        self.itemsIncluded = set()

    def __lt__(self, other):
        """ Compare nodes based on their bound values. """
        return self.bound < other.bound

    def __gt__(self, other):
        """ Compare nodes based on their bound values. """
        return self.bound > other.bound

    def __ge__(self, other):
        """ Compare nodes based on their bound values. """
        return self.bound >= other.bound

    def __str__(self):
        return str(self.bound)

def bound(u):
    """
    Calculates the bound on the best possible profit for node u by using what is already in the knapsack and running the fractional knapsack calculation on the remaining items
    """
    if(u.weight >= W_knapsack_limit):
        return 0
    else:
        result = u.profit
        j = u.level + 1
        totweight = u.weight
        while({j <= n} and (totweight + w[j] <= W_knapsack_limit)):
            totweight = totweight + w[j]
            result = result + p[j]
            j += 1
        k = j
        if (k<=n):
            result = result + (W_knapsack_limit - totweight) * (p[k]/w[k])
        return result

def knapsack(n, p, w, W_knapsack_limit):
    PQ = MaxHeap()
    v = Node(0,0,0,0)
    maxprofit = 0
    maxnode = v
v.bound = bound(v)
print( "Enqueuing: Level: %d, Weight: %d, Profit: %d, Bound: %d" %
      (v.level, v.weight, v.profit, v.bound))
PQ.add(v)
while not PQ.isEmpty():
    v = PQ.pop()
    print( "Dequeuing: Level: %d, Weight: %d, Profit: %d, Bound: %d" %
           (v.level, v.weight, v.profit, v.bound), end="")
    print(v.itemsIncluded)
    if v.bound > maxprofit:
        u = Node(0, 0, 0, 0)
        u.level = v.level + 1
        u.weight = v.weight + w[u.level]
        u.profit = v.profit + p[u.level]
        u.itemsIncluded = set(v.itemsIncluded)
        u.itemsIncluded.add(u.level)
        if u.weight <= W_knapsack_limit and u.profit > maxprofit:
            maxprofit = u.profit
            maxnode = u
        u.bound = bound(u)
    if u.bound > maxprofit:
        print( "    Enqueuing: Level: %d, Weight: %d, Profit: %d, Bound: %d" %
               (u.level, u.weight, u.profit, u.bound), end="")
        print(u.itemsIncluded)
        PQ.add(u)
        u = Node(0, 0, 0, 0)
        u.level = v.level + 1
        u.weight = v.weight
        u.profit = v.profit
        u.bound = bound(u)
        u.itemsIncluded = set(v.itemsIncluded)
        if u.bound > maxprofit:
            PQ.add(u)
            print( "    Enqueuing: Level: %d, Weight: %d, Profit: %d, Bound: %d" %
                   (u.level, u.weight, u.profit, u.bound), end="")
            print(u.itemsIncluded)
return maxprofit, maxnode.itemsIncluded

class MaxHeap(object):
    
    """ Used to implement a priority queue as a Max. Binary Heap. """
    
    def __init__(self):
        self._heap = []

    def __len__(self):
        return len(self._heap)

    def isEmpty(self):
        return len(self) == 0

    def peek(self):
        if self.isEmpty():
            raise Exception( "Heap is empty")
        return self._heap[0]

    def add(self, item):
        self._heap.append(item)
        curPos = len(self._heap) - 1
        while curPos > 0:
            parent = (curPos - 1) // 2
            parentItem = self._heap[parent]
if parentItem >= item:
    break
else:
    self._heap[curPos] = self._heap[parent]
    self._heap[parent] = item
    curPos = parent

def pop(self):
    if self.isEmpty():
        raise Exception( "Heap is empty" )

    topItem = self._heap[0]
    bottomItem = self._heap.pop(len(self._heap) - 1)
    if len(self._heap) == 0:
        return bottomItem

    self._heap[0] = bottomItem
    lastIndex = len(self._heap) - 1
    curPos = 0
    while True:
        leftChild = 2 * curPos + 1
        rightChild = 2 * curPos + 2
        if leftChild > lastIndex:
            break
        if rightChild > lastIndex:
            maxChild = leftChild;
        else:
            leftItem = self._heap[leftChild]
            rightItem = self._heap[rightChild]
            if leftItem > rightItem:
                maxChild = leftChild
            else:
                maxChild = rightChild
            maxItem = self._heap[maxChild]
        if bottomItem >= maxItem:
            break
        else:
            self._heap[curPos] = self._heap[maxChild]
            self._heap[maxChild] = bottomItem
            curPos = maxChild
    return topItem

def __iter__(self):
    tempList = []
    for item in self._heap:
        tempList.append(item)
    resultList = []
    while not self.isEmpty():
        resultList.append(self.pop())
    self._heap = tempList
    return iter(resultList)

def __str__(self):
    def strHelper(position, level):
        result = ""
        if position < len(self):
            result += strHelper(2 * position + 2, level + 1)
            result += "|" * level
            result += str(self._heap[position]) + "\n"
            result += strHelper(2 * position + 1, level + 1)
        return result
    return strHelper(0, 0)
# Global Problem Instance Information
n = 5
p = [-1, 20, 30, 35, 12, 3]
w = [-1, 2, 5, 7, 3, 1]
W_knapsack_limit = 13
maxprofit = 0
maxprofit, itemsIncluded = knapsack(n, p, w, W_knapsack_limit)
print("maxprofit: \%d" \% maxprofit)
print("Items to Steal: ", str(itemsIncluded))
IDLE tmp wukfit
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================================= RESTART =================================================
>>> Enqueuing: Level: 0, Weight: 0, Profit: 0, Bound: 80
  Enqueuing: Level: 1, Weight: 2, Profit: 20, Bound: 80[1]
  Enqueuing: Level: 1, Weight: 0, Profit: 0, Bound: 69set()
Dequeuing: Level: 1, Weight: 2, Profit: 20, Bound: 80[1]
  Enqueuing: Level: 2, Weight: 7, Profit: 50, Bound: 80[1, 2]
  Enqueuing: Level: 2, Weight: 2, Profit: 20, Bound: 70[1]
Dequeuing: Level: 2, Weight: 7, Profit: 50, Bound: 80[1, 2]
  Enqueuing: Level: 3, Weight: 7, Profit: 50, Bound: 65[1, 2]
Dequeuing: Level: 2, Weight: 2, Profit: 20, Bound: 70[1]
  Enqueuing: Level: 3, Weight: 9, Profit: 55, Bound: 70[1, 3]
Dequeuing: Level: 3, Weight: 9, Profit: 55, Bound: 70[1, 3]
  Enqueuing: Level: 4, Weight: 12, Profit: 67, Bound: 70[1, 3, 4]
Dequeuing: Level: 4, Weight: 12, Profit: 67, Bound: 70[1, 3, 4]
Dequeuing: Level: 1, Weight: 0, Profit: 0, Bound: 69set()
Dequeuing: Level: 3, Weight: 7, Profit: 50, Bound: 65[1, 2]
maxprofit: 70
Items to Steal: {1, 3, 4, 5}
>>>
class Node(object):
    """ Node objects hold state information about for each node in the TSP problem"""
    def __init__(self, level, path, distance):
        self.level = level
        self.path = path
        self.distance = distance
        self.bound = 0

    def __lt__(self, other):
        """ Compare nodes based on their bound values. """
        return self.bound < other.bound

    def __gt__(self, other):
        """ Compare nodes based on their bound values. """
        return self.bound > other.bound

    def __ge__(self, other):
        """ Compare nodes based on their bound values. """
        return self.bound >= other.bound

    def __le__(self, other):
        """ Compare nodes based on their bound values. """
        return self.bound <= other.bound

    def __str__(self):
        return '{'+str(self.bound)+" path:"+str(self.path)+'}'

def findLastVertex(n, u):
    for item in range(1, n+1):
        if not (item in u.path):
            return item
    raise Exception("In findLastVertex, but none found...")

def bound(u):
    """ Calculates the bound on the best possible completion of the tour by using the minimum edge leaving the vertex and going to some possible vertex (e.g., vertex not already on the path or back to v1)"""

dist = u.distance
verticesOnPartialTour = set(u.path)
verticesNotOnPartialTour = set(range(1,n+1)) - verticesOnPartialTour
lastVertexOnPartialTour = u.path[-1]
minEdge = 999999999
for possibleToVertex in verticesNotOnPartialTour:
    if W[lastVertexOnPartialTour][possibleToVertex] > 0 and
    W[lastVertexOnPartialTour][possibleToVertex] < minEdge:
        minEdge = W[lastVertexOnPartialTour][possibleToVertex]
dist += minEdge

possibleToVerticesSet = verticesNotOnPartialTour.copy()
possibleToVerticesSet.add(1)
for fromVertex in verticesNotOnPartialTour:
    minEdge = 9999999999
    for possibleToVertex in possibleToVerticesSet:
        if W[fromVertex][possibleToVertex] > 0 and
            W[fromVertex][possibleToVertex] < minEdge:
            minEdge = W[fromVertex][possibleToVertex]
    dist += minEdge

return dist

def TSP(n, W):
    global tourFound
    global bestTour
    global minlength
    PQ = MinHeap()
    v = Node(0, [1], 0)

    v.bound = bound(v)
    print( "Enqueueing: Level: %d, distance: %d, Bound: %d" % (v.level,
        v.distance, v.bound), end="\n")
    print( ", Path: ", v.path)
    PQ.add( v)
    while not PQ.isEmpty():

        v = PQ.pop()
        print( "Dequeueing: Level: %d, distance: %d, Bound: %d" %
            (v.level, v.distance, v.bound), end="\n")
        print( ", Path: ", v.path)
        if not tourFound or v.bound < minlength:
            for i in range(2, n+1):
                if i in v.path or W[v.path[-1]][i] <= 0:
                    continue
                u = Node(0, [], 0)
                u.level = v.level + 1
                u.path = v.path + [i]
                u.distance = v.distance + W[v.path[-1]][i]
                if u.level == n-2:
                    lastVertex = findLastVertex(n, u)
                    if W[i][lastVertex] > 0 and W[lastVertex][1] > 0:
                        u.distance += W[i][lastVertex] + W[lastVertex][1]
                        u.path.append(lastVertex)
                        u.path.append(1)
                        tourFound = True
                        if u.distance < minlength:
                            print("NEW BEST TOUR:", u.distance, u.path)
                            bestTour = u.path
                            minlength = u.distance
                    else:
                        continue

        u.bound = bound(u)
        if not tourFound or u.bound < minlength:
            print( "Enqueueing: Level: %d, distance: %d, Bound: %d" %
                (u.level, u.distance, u.bound), end="\n")
            print( ", Path: ", u.path)
            PQ.add(u)

    return minlength, bestTour

class MinHeap(object):
def __init__(self):
    self._heap = []

def __len__(self):
    return len(self._heap)

def isEmpty(self):
    return len(self) == 0

def peek(self):
    if self.isEmpty():
        raise Exception( "Heap is empty" )
    return self._heap[0]

def add(self, item):
    self._heap.append(item)
    curPos = len(self._heap) - 1
    while curPos > 0:
        parent = (curPos - 1) // 2
        parentItem = self._heap[parent]
        if parentItem <= item:
            break
        else:
            self._heap[curPos] = self._heap[parent]
            self._heap[parent] = item
            curPos = parent

def pop(self):
    if self.isEmpty():
        raise Exception( "Heap is empty" )

    topItem = self._heap[0]
    bottomItem = self._heap.pop()
    if len(self._heap) == 0:
        return bottomItem

    self._heap[0] = bottomItem
    lastIndex = len(self._heap) - 1
    curPos = 0
    while True:
        leftChild = 2 * curPos + 1
        rightChild = 2 * curPos + 2
        if leftChild > lastIndex:
            break
        if rightChild > lastIndex:
            minChild = leftChild;
        else:
            leftItem = self._heap[leftChild]
            rightItem = self._heap[rightChild]
            if leftItem < rightItem:
                minChild = leftChild
            else:
                minChild = rightChild
        minItem = self._heap[minChild]
        if bottomItem <= minItem:
            break
        else:
            self._heap[curPos] = self._heap[minChild]
            self._heap[minChild] = bottomItem
            curPos = minChild
return topItem

def __iter__(self):
    tempList = []
    for item in self._heap:
        tempList.append(item)
    resultList = []
    while not self.isEmpty():
        resultList.append(self.pop())
    self._heap = tempList
    return iter(resultList)

def __str__(self):
    def strHelper(position, level):
        result = ""
        if position < len(self):
            result += strHelper(2 * position + 2, level + 1)
            result += "|" * level
            result += str(self._heap[position]) + "\n"
            result += strHelper(2 * position + 1, level + 1)
        return result
    return strHelper(0, 0)

# Global Problem Instance Information
n = 8
W = 
[-1,-1,-1, 5, 8,-1,-1,-1,-1,-1],
[-1,-1,-1, 4,-1, 4,-1,-1,-1,-1],
[-1,-1,-1,-1, 2,-1,-1, 5,-1],
[-1,-1,-1,-1,-1,-1,-1,-1,-1,-1],
[-1,-1,-1, 1,-1,-1,-1,-1,-1,-1],
[-1,-1, 6,-1,-1, 2,-1,-1,-1,-1],
[-1,-1,-1,-1, 3,-1, 8,-1,-1,-1],
[-1,-1,-1,-1,-1,-1, 5, 4,-1]]
tourFound = False
bestTour = []
minlength = 99999999999
minlength, optimalTour = TSP(n,W)
print("minlength: %d" % minlength)
print("Optimal Tour: ", optimalTour)
Enqueuing: Level: 0, distance: 0, Bound: 28, Path: [1]
Dequeuing: Level: 0, distance: 0, Bound: 28, Path: [1]
Enqueuing: Level: 1, distance: 5, Bound: 28, Path: [1, 2]
Enqueuing: Level: 1, distance: 8, Bound: 31, Path: [1, 3]
Dequeuing: Level: 1, distance: 5, Bound: 28, Path: [1, 2]
Enqueuing: Level: 2, distance: 9, Bound: 28, Path: [1, 2, 3]
Enqueuing: Level: 2, distance: 9, Bound: 28, Path: [1, 2, 3]
Enqueuing: Level: 2, distance: 9, Bound: 28, Path: [1, 2, 3]
Enqueuing: Level: 3, distance: 11, Bound: 33, Path: [1, 2, 3, 4]
Enqueuing: Level: 3, distance: 14, Bound: 32, Path: [1, 2, 3, 7]
Dequeuing: Level: 1, distance: 8, Bound: 31, Path: [1, 3]
Enqueuing: Level: 2, distance: 10, Bound: 36, Path: [1, 3, 4]
Enqueuing: Level: 2, distance: 13, Bound: 35, Path: [1, 3, 7]
Dequeuing: Level: 3, distance: 14, Bound: 32, Path: [1, 2, 3, 7]
Enqueuing: Level: 4, distance: 17, Bound: 32, Path: [1, 2, 3, 7, 4]
Enqueuing: Level: 4, distance: 22, Bound: 10000000031, Path: [1, 2, 3, 7, 6]
Dequeuing: Level: 4, distance: 22, Bound: 10000000031, Path: [1, 2, 3, 7, 6]
Dequeuing: Level: 5, distance: 24, Bound: 32, Path: [1, 2, 3, 7, 4, 8]
Dequeuing: Level: 5, distance: 24, Bound: 32, Path: [1, 2, 3, 7, 4, 8]

NEW BEST TOUR: 32 [1, 2, 3, 7, 4, 8, 6, 5, 1]
import math

class Node:
    def __init__(self, key, link=None):
        self.key = key
        self.link = link

class digitList:
    def __init__(self):
        self.start = None
        self.end = None

def radixSort(masterList, numDigits):
    p = masterList
    print('initial masterList**********************************')
    while p != None:
        print(p.key, end=' ')  
        p = p.link
    print()
    for i in range(1, numDigits+1):
        digitLists = distribute(masterList, i, numDigits)
        masterList = coalesce(digitLists)
        p = masterList
        print('after radixSort loop
i=%d,**********************************
' % i)
        while p != None:
            print(p.key, end=' ')  
            p = p.link
        print()

    return masterList

def distribute(masterList, i, numDigits):
    digitLists = []
    for digit in range(10):
        digitLists.append(digitList())

    p = masterList
    while p != None:
        j = int('0'*numDigits+str(p.key))[-i]
        if digitLists[j].start == None:
            digitLists[j].start = p
            digitLists[j].end = p
        else:
            digitLists[j].end.link = p
            digitLists[j].end = p
        temp = p
        p = p.link
    temp.link = None

    return digitLists

def coalesce(digitLists):
    masterList = None
    j = 0
    masterListEnd = None
    while j <= 9:
if digitLists[j].start != None:
    if masterList == None:
        masterList = digitLists[j].start
        masterListEnd = digitLists[j].end
    else:
        masterListEnd.link = digitLists[j].start
        masterListEnd = digitLists[j].end
j += 1
return masterList

def main(myList):
    masterListStart = None
    masterListEnd = None
    if len(myList) > 0:
        masterListStart = Node(myList[0])
        masterListEnd = masterListStart
        maxNum = myList[0]
    for index in range(1, len(myList)):
        masterListEnd.link = Node(myList[index])
        masterListEnd = masterListEnd.link
        if myList[index] > maxNum:
            maxNum = myList[index]
    numDigits = math.floor(math.log10(maxNum)) + 1
    masterListStart = radixSort(masterListStart, numDigits)
    for index in range(len(myList)):
        myList[index] = masterListStart.key
        masterListStart = masterListStart.link

aList = [492, 803, 341, 739, 604, 129, 358, 759, 34, 444, 197, 390]
print('unsorted list:', aList)
main(aList)
print(' sorted list:', aList)
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ==============================================================
>>> unsorted list: [492, 803, 341, 739, 604, 129, 358, 759, 34, 444, 197, 390]
initial masterList============================================================
492 803 341 739 604 129 358 759 34 444 197 390
after radixSort loop i= 1 ==================================================
390 341 492 803 604 34 444 197 358 739 129 759
after radixSort loop i= 2 ==================================================
803 604 129 34 739 341 444 358 759 390 492 197
after radixSort loop i= 3 ==================================================
34 129 197 341 358 390 444 492 604 739 759 803
sorted list: [34, 129, 197, 341, 358, 390, 444, 492, 604, 739, 759, 803]
>>>