CS532 Homework 5

Archana Machireddy

Question 1

In b(i,j,w), i and j are the indices referring to a start and end position, and w is the maximum allowed weight. The function gives the best value for items i through j, having weight at most w.

Question 2

The base case is whether or not to include the first item, which has index 0. If the weight of the first item(w_0) is more than the maximum allowed weight(w), it cannot be included, and therefore the best value for items having weight at most w is still 0. If the weight of the first item is less than or equal to the maximum allowed weight, it can be included, and therefore the best value for items having weight at most w (considering only first item) is now equal to the value of the first item v_0.

Question 3

In order to obtain the best subset from a list on n items, for each item we have to make the choice of including or excluding it. We have to choose the case that gives a higher overall value for that subset. If there are n items, and R is the subset of items resulting in best value having weight at most w, if item i is removed from this subset, the remaining items must be the best set of items weighting at most $w – w_i$ that can be picked from the original n-1 items, excluding item i.

The value of best subset of items (S_i) that has total weight w is either:

1. The value of best subset of S_{i-1} that has total weight w (first term: b(0,i-1,w)) (i.e. exclude the item), or
2. The value of best subset of S_{i-1} that has total weight $w – w_i$ plus the value of item i v_i (second term: b(0,i-1, $w – w_i$)+ v_i) (i.e. include the item)

We take the max as we want the best overall value for the items. If including the item gives a higher overall value, we go with second choice, else we exclude the item.

Question 4

At each step in recursion there are two choices, include the item or exclude the item from the list of best combination of items. For each choice there is one sub-problem to solve.

Question 5

The two core requirements of dynamic programming are optimal substructure and overlapping sub-problems. A problem exhibits optimal substructure, if an optimal solution to the problem contains within it the optimal solution to its sub-problems. This problem exhibits the optimal sub-structure property. As stated previously, ff there are n items, and S is the subset of items resulting in best value having weight at most w, if item i is removed from this subset, the remaining items must be the best set of items weighting at most $w - w_i$ that can be picked from the original n-1 items, excluding item i.

While solving for subsets of the original items, we calculate the best value for a few subsets repeatedly; this gives rise to overlapping sub-problems. Therefore, 0-1 knapsack problem can be solved using dynamic programming.

Question 6

The sub-problems need to be ordered. In bottom-up we solve smaller problems and use the results of the smaller problems to compute results of bigger problems. Here for each item i, we use the values calculated excluding that item for different weights and these values would have already been filled by the time we go to item i. So we need to solve the smaller problems first to easily calculate solutions for bigger problems.

Question 7

Nested list is a simple data structure to store the intermediate results. Code to initialize it to all zeros:

```
cost = [[0 for i in range(w+1)] for j in V]
```

Question 8

```
def knapsack1(W,V,w):
    cost = [[0 for i in range(w+1)] for j in V]
```

```
    for i in range(len(V)):
        for j in range(w+1):
            if i == 0:
                if W[i] <= j:
                    cost[i][j] = V[i]
                else:
                    cost[i][j] = 0
            elif W[i] <= j:
                cost[i][j] = max(cost[i-1][j],cost[i-1][j-W[i]]+V[i])
            else:
                cost[i][j] = cost[i-1][j]
    return cost[i][j]
```

Question 8 Critique

I was returning cost[i][j] as in my code by the end they would have reach the final cell. But specifying the final cell explicitly is better `(cost[len(V)-1][w])`. I put the i == 0 case inside the loop. I should have left it out side the loop as we are just going over it once, instead of it checking it now for every i and j.

```
def knapsack1(W,V,w):
    cost = [[0 for i in range(w+1)] for j in V]
    for j in range(w+1):
        cost[0][j] = 0
        if W[0] <= j:
            cost[0][j] = V[0]
        for i in range(1,len(V)):
            cost[i][j] = cost[i-1][j]
            if W[i] <= j:
                cost_inc = cost[i-1][j-W[i]]+V[i]
                if cost_inc > cost[i][j]:
                    cost[i][j] = cost_inc
    return cost[len(V)-1][w]
```

Question 9

Solution 1: Backtracking on the calculated cost matrix

```
def knapsack2(W,V,w): #1
    cost = [[0 for i in range(w+1)] for j in V]
    for i in range(len(V)):
        for j in range(w+1):
            if i == 0:
                if W[i] <= j:
                    cost[i][j] = V[i]
```

```
                else:
                    cost[i][j] = 0
            else:
                cost[i][j] = cost[i-1][j]
                if W[i] <= j:
                    cost_inc = cost[i-1][j-W[i]]+V[i]
                    if cost_inc > cost[i][j]:
                        cost[i][j] = cost_inc


    cost_final = cost[len(V)-1][w]
    items = []
    cur_w = w
    for i in range(len(V),-1,-1):
        if i == 0:
            if cur_w >= W[i]:
                items.append(i)
            break
        if cost_final != cost[i-1][cur_w]:
            items.append(i)
            cost_final = cost_final - V[i]
            cur_w = cur_w - W[i]
    return (cost[len(V)-1][w],items)
```

Solution 2: Using a nested list to store if the item is included in the best solution, and backtracking on this list

```
def knapsack2(W,V,w):#2
    cost = [[0 for i in range(w+1)] for j in V]
    items = [[0 for i in range(w+1)] for j in V]
    for i in range(len(V)):
        for j in range(w+1):
            if i == 0:
                if W[i] <= j:
                    cost[i][j] = V[i]
                    items[i][j] = 1
                else:
                    cost[i][j] = 0
            else:
                cost[i][j] = cost[i-1][j]
                if W[i] <= j:
                    cost_inc = cost[i-1][j-W[i]]+V[i]
                    if cost_inc > cost[i][j]:
                        cost[i][j] = cost_inc
                        items[i][j] = 1
    cost_final = cost[len(V)-1][w]
    final_items = []
    cur_w = w
    for i in range(len(V)-1,-1,-1):
```

```
        if items[i][cur_w] == 1:
            final_items.append(i)
            cur_w = cur_w - W[i]
    return (cost[len(V)-1][w],final_items)
```

Question 10

```
def knapsack3_sub(W,V,w): # Core top down code
    n = len(V)-1
    if cache[(n,w)][0] != -1:
        return cache[(n,w)]

    if n == 0:
        if W[n] <= w:
            cache[(n,w)] = (V[n],0)
            return cache[(n,w)]
        else:
            cache[(n,w)] = (0,0)
            return cache[(n,w)]
    if W[n] > w:
        cache[(n,w)] = (knapsack3(W[:n],V[:n],w),0)
        return cache[(n,w)]
    else:
        x1 = knapsack3(W[:n],V[:n],w)[0]
        x2 = V[n] + knapsack3(W[:n],V[:n],w-W[n])[0]
        if x2 > x1:
            cache[(n,w)] = (x2,1)
        else:
            cache[(n,w)] = (x1,0)
        return cache[(n,w)]


def knapsack3(W,V,w): # Code to return best value and list of indices
of items included
    best,seq = knapsack3_sub(W,V,w)
    cost_final = cache[len(V)-1,w][0]
    final_items = []
    cur_w = w
    for i in range(len(V)-1,-1,-1):
        if cache[i,cur_w][1] == 1:
            final_items.append(i)
            cur_w = cur_w - W[i]
    return (best,final_items)
```

```
Archanas-MBP:HW5 archana$ python hw5.py
('Version 1:', 'Best value: ', 220)
('Version 2:', 'Best value: ', 220, 'List index of items: ', [2, 1])
('Version 3:', 'Best value: ', 220, 'List index of items: ', [2, 1])
Archanas-MBP:HW5 archana$
```

Question 11

```python
def parent(self,i):
    return int((i-1)//2)

def left(self,i):
    return 2*i+1

def right(self,i):
    return 2*i+2
```

Question 12

```python
def min_heapify(self,i):
    l = self.left(i)
    r = self.right(i)
    if l <= self.heap_size-1 and self.A[l] < self.A[i]:
        smallest = l
    else:
        smallest = i
    if r <= self.heap_size-1 and self.A[r] < self.A[smallest]:
        smallest = r
    if smallest != i:
        w = self.A[i]
        self.A[i] = self.A[smallest]
        self.A[smallest] = w
        self.min_heapify(smallest)

def build_min_heap(self):
    self.heap_size = self.length
    for i in range((self.length//2)-1,-1,-1):
        self.min_heapify(i)

def heap_extract_min(self):
```

```
    if self.heap_size < 1:
        print("heap overflow")
    min = self.A[0]
    self.A[0] = self.A[self.heap_size-1]
    del self.A[self.heap_size-1]
    self.heap_size = self.heap_size-1
    self.min_heapify(0)
    return min
```

Code to test the above functions:

```
b=[4,5,1,8,9,7,10]
h1=heap(b)
print('Input heap: ', b)
h1.min_heapify(0)
print('min_heapify(0) result: ',b)

print('-'*80)

b=[10, 8, 9, 7, 6, 5, 4]
h1=heap(b)
print('Input heap: ', b)
h1.build_min_heap()
print('Build_min_heap result: ',b)
m=h1.heap_extract_min()
print('Extract min: ',m)

print('-'*80)

b=[10, 8, 9, 7, 6, 5, 4]
h1=heap(b)
print('Input heap: ', b)
h1.build_min_heap_iterative()
print('Build_min_heap_iterative result: ',b)

h1.min_heap_insert(1)
print('Heap after inserting 1: ',b)
```

```
Archanas-MBP:HW5 archana$ python hw5.py
('Input heap: ', [4, 5, 1, 8, 9, 7, 10])
('min_heapify(0) result: ', [1, 5, 4, 8, 9, 7, 10])
--------------------------------------------------------------------------------
('Input heap: ', [10, 8, 9, 7, 6, 5, 4])
('Build_min_heap result: ', [4, 6, 5, 7, 8, 10, 9])
('Extract min: ', 4)
--------------------------------------------------------------------------------
('Input heap: ', [10, 8, 9, 7, 6, 5, 4])
('Build_min_heap_iterative result: ', [4, 6, 5, 7, 8, 10, 9])
('Heap after inserting 1: ', [1, 4, 5, 6, 8, 10, 9, 7])
Archanas-MBP:HW5 archana$
```

Question 13

```python
def min_heapify_iterative(self,i):
    while i <= self.heap_size:
        l = self.left(i)
        r = self.right(i)
        if  l <= self.heap_size-1 and self.A[l] < self.A[i]:
            smallest = l
        else:
            smallest = i
        if r <= self.heap_size-1 and self.A[r] < self.A[smallest]:
            smallest = r
        if smallest != i:
            w = self.A[i]
            self.A[i] = self.A[smallest]
            self.A[smallest] = w
            i = smallest
        else:
            break
```

Question 14

```python
def min_heap_insert(self,key):
    self.heap_size = self.heap_size + 1
    self.A.append(-float('Inf'))
    if key < self.A[self.heap_size-1]:
        print('New key smaller than current key')
    self.A[self.heap_size-1] = key
    i = self.heap_size-1
    while i > 0 and self.A[self.parent(i)] > self.A[i]:
        w = self.A[i]
```

```
self.A[i] = self.A[self.parent(i)]
self.A[self.parent(i)] = w
i = self.parent(i)
```