Daniel Oliveros A02093272
April 3, 2018

Assignment 5: Dynamic Programming

Problem #1 unlimited knapsack

1. **Algorithm Description:** For this algorithm, we consider an extra subproblem than the one in the basic knapsack. Not only do we check if the problem can be solved by not adding the number, or if the table can be solved by adding the number to the solution found in the previous column, but we also check if we can add the number in our column to elements found further down in it. The result is that we can add an unlimited number of copies of each element to the solutions found in each column.

2. **Pseudocode:**

```
def problem1(a, M):
    #create basic table
    n = [[0]*(len(a)+1) for i in range(1,M+2)]
    n[0] = [1]*(len(a)+1)
    #iterate through table
    for i in range(1, len(a)+1): #O(n)
        for j in range(1, M+1): #O(M)
            if(j<a[i-1]):
                #check subproblem available
                n[j][i] = n[j][i-1]
            else:
                #check all subproblems
                n[j][i]=max(n[j][i-1],n[j-a[i-1]][i-1], n[j-a[i-1]][i])
    #check if we could fill a knapsack of size M
    return (n[M][len(a)] == 1)
#testing code
a = [2, 7, 9, 3]
k = 14
print(problem1(a, k))
```

3. **Correctness:** For any given number, we need to check if we can fill a knapsack of that size by all means allowed. Since in this scenario we can also add items to our knapsack from the column we're in, then by checking if this is a plausible solution we satisfy this requirement.

4. **Time Analysis:** Since we perform constant operations for each element in our table, and our table has a size of $n * M$, then our total runtime for this function will be $O(nM)$

Problem #2 subset closest to M

1. **Algorithm Description:** For this problem, we create a table of size $n * K$, where n is the number of elements in our array, and K is the sum of all elements in our array. Solving a knapsack problem on this table gives us every single knapsack our set of elements could fill up. From here, we check at location M and see if it's a problem we could solve, if so, we return True, otherwise, we check all locations closest to it until we find a solvable knapsack and return that M

2. **Pseudocode:**

```
def problem2(a, M):
    k = sum(a)
    n = [[0]*(len(a)+1) for i in range(1,k+2)]
    n[0] = [1]*(len(a)+1)
    #populate table with all solutions
    for i in range(1, len(a)+1):#O(n)
        for j in range(1, k+1):#O(k)
            if(j<a[i-1]):
                n[j][i] = n[j][i-1]
            else:
                n[j][i]=max(n[j][i-1],n[j-a[i-1]][i-1])
    #check over and under M to find closest solution
    for i in range (0, k):#O(k)
        if(0<= M + i <= k):
            if(n[M+i][len(a)] == 1):
                return(M+i)
        if(0<= M - i <= k):
            if(n[M-i][len(a)] == 1):
                return(M-i)
#testing code
M = 15
a = [1, 4, 7, 12]
print(problem2(a, M))
```

3. **Correctness:** By "oscillating" around the location M of our array, we can make sure that we get the closest solution available to that value.

4. **Time Analysis:** Since we perform constant operations per each entry on a table with $n * K$ entries, the runtime for that part is $O(nK)$. The runtime for checking the closest solution to M is just K, since in the worst case scenario we'll end up checking the whole column. So, the total runtime for our array is $O(n * K + k) \Rightarrow O(nK)$

Problem #3 Knapsack maximizing for value

1. **Algorithm Description:** This problem works similarly to the original knapsack, except that we make sure that we fill each entry with the value of the highest solution that can exist for it. We also keep a variable storing the maximum value of our table, so whenever we fill out an entry of our table, we compare it to that variable and update it if necessary.

2. **Pseudocode:**

```
def problem3(a, k):
    #create basic table
    n = [[0]*(len(a)+1) for i in range(0,k+1)]
    n[0] = [1]*(len(a)+1)
    mx = 0
    #populate table based on weights of values stored
    for i in range(1, len(a)+1):
        for j in range(1, k+1):
            #avoid checking locations of array you shouldn't access
            if(j<a[i-1][0]):
                n[j][i] = n[j][i-1]
            else:
                #determine if value needs to be added
                val=max(n[j][i-1],n[j-a[i-1][0]][i-1])
                if(val == 0):
                    n[j][i] = val
                #if value can be added, determine which way of adding it
                    ↪ will give the greatest value
                else:
                    if(j==a[i-1][0]):
                        val = max(n[j][i-1],n[j-a[i-1][0]][i-1]+a[i
                            ↪ -1][1]-1)
                        n[j][i] = val
                    else:
                        val = max(n[j][i-1],n[j-a[i-1][0]][i-1]+a[i-1][1])
                        n[j][i] = val
                #check if value is greater than maximum value kept
                if(val>mx):
                    mx = val
    return mx
#testing code
a = [[2,5], [4,7], [7,9], [12,17], [7,11]]
k = 14
print(problem3(a, k))
```

3. **Correctness:** Since the problem asks us to report the subset with the greatest value bound by the size of our knapsack, we fill out all possible entries of this knapsack while ensuring each entry has the maximum value it can have for it.

4. **Time Analysis:** Since we fill out a table of size $n * M$, and perform simple constant operations for each entry, our total runtime for this algorithm is $O(nM)$

Problem #4 longest monotonically increasing subsequence

1. **Algorithm Description:** In this setup, the subproblem for any element is whether or not it's greater than the elements that came before it. Each element stores the length of the longest monotonically increasing subsequence that ends at that element, so when we determine an element is greater than a set of elements that came before it, we get the greatest value of those elements and store its value plus one for our element's entry.

2. **Pseudocode:**

```
def problem4(a):
    #create basic array
    n = len(a)
    t = [0]*n
    #go through each element
    for i in range (0, n):
        mono = 1
        #check elements that came before a given element
        for j in range (0, i):
            #if an element that came before is smaller than my element,
                ↪ check the value of its subsequence
            if(a[i]>a[j]):
                mono = max(mono, t[j]+1)
        t[i] = mono
    #return longest subsequence
    longest = max(t)
    return(longest)
#testing code
a = [20, 5, 14, 8, 10, 3, 12, 7, 16]
print(problem4(a))
```

3. **Correctness:** By checking every element that came before a given element, we ensure we're able to check all possible scenarios. The length of the greatest subsequence at a given entry is equal to the max of the subsequences it's can be appended to plus one.

4. **Time Analysis:** For each entry, the number of operations executed for it is equal to the number of entries that came before it. So, the total runtime for our algorithm is $\sum_{i=1}^{n-1} i - 1 \Rightarrow \frac{(n-1)(n-2)}{2}$. The asymptotic runtime for this algorithm is $O(n^2)$

People I worked with for this assignment:

- Olivia Garrido
- Chris Darais
- Erik Davtyan