Muhammed ÖZKAN 151044084

CSE-321 Introduction to Algorithm Design

Fall 2020 Homework 5

1.

For this question, instead of a bottom-up approach that is generally used in the subset-sum problem, I utilized recursive which we have seen in the calculation of Fibonacci numbers. This method is used whenever there are multiples of the same operations that needs to be calculated repeatedly. In those situations, saving previously calculated variables in a table comes in handy. This problem of finding a subset with a specific sum can be represented as the 0-1 Knapsack Problem. We are still going to construct a recursive structure where we either include a number or not which creates two branches to the call tree. But adding an operation that saves the element and the sum with its inclusion to a table, we don't need to calculate it repeatedly and can just check the table for a quick look-up. The worst-case running time of this algorithm for any S = sum value is **$O(2^n)$.**

We recursively generate all subsets. We keep track of elements of current subset. If sum of elements in current subset becomes equal to given sum, we print the subset.

```
def find_subset(array, n, table, sum) :

   if (sum == 0 and len(table)!=0):
      for value in table :
         print(value, end=" ")
      print("")
      return

   if (n == 0):
      return

   find_subset(array, n - 1, table, sum)
   temp = [] + table
   temp.append(array[n - 1])
   find_subset(array, n - 1, temp, sum - array[n - 1])
```

Numbers: [2, 3, -5, -8, 6, -1]
-5 3 2
6 -8 2
-1 6 -5
-1 6 -8 3

2.

We start from the bottom up from the bottom row of nodes. The minimum path sum for these nodes is the values of the nodes themselves. And after that, the minimum path sum at node i of row j will be the minimum path sum of its two children + the value of the node, i.e .:

triangle

[**2**],
[**5**,4],
[**1**,4,7],
[8,**6**,9,6]

table

14, 0, 0, 0
12, 14, 0, 0
7, 10, 13, 0
8, **6**, 9, 6

If we look closely then we can see that the table has meaningful values in lower half only and at each level bottom up we have one of the column value getting fixed. So, we could have basically used the bottom level array as the dp table and at each level we update the columns bottom up. The worst-case running time of this algorithm for any n*n matrices is **$O(n^2)$.**

```
def minSumPath(triangle):

   table = [0] * len(triangle)
   n = len(triangle) - 1

   for i in range(len(triangle[n])):
      table[i] = triangle[n][i]

   for i in range(len(triangle) - 2, -1,-1):
      for j in range( len(triangle[i])):
         table[j] = triangle[i][j] + min(table[j],table[j + 1]);

   return table[0]
```

3.

$$F(W) = max_{j:W \geq w_j} \{F(W - w_j) + v_j\}$$

$$F(W) = 0 \; if \; W < w_j, \forall j \in [1,2, \ldots., n]$$

Using this recurrence, the algorithm can fill the one-row table in the same way as the change making problem.

(Remember that the recurrence for change making problem was $F(n) = min\{F(n - d_i) + 1\}.$)

F(0)=0,
F(1)=0,
F(2)=max{F(2-2)+3}=3,
F(3)=max{F(3-2)+3}=3,
F(4)=max{F(4-2)+3, F(4-4)+4}=6,
F(5)=max{F(5-2)+3, F(5-4)+4, F(5-5)+10}=10,
F(6)=max{F(6-2)+3, F(6-4)+4, F(6-5)+10}=10,
F(7)=max{F(7-2)+3, F(7-4)+4, F(7-5)+10}=13,
F(8)=max{F(8-2)+3, F(8-4)+4, F(8-5)+10}=13,
F(9)=max{F(9-2)+3, F(9-4)+4, F(9-5)+10}=16.

We can find selected items by backtracking.

When finding F(9), we use F(9-5)+10 as the maximum. So we use item 1.

When finding F(4) we use F(4-2)+3, so we use item 3.

Similarly, when finding F(2) we use F(2-2)+3, so we use item 3. We use item 3 twice, and item 1 once. Time Complexity **O(W.n)**

```
def Knapsack(W, wt, val,n):

   table = [0 for i in range(W + 1)]

   ans = 0

   for i in range(W + 1):
      for j in range(n):
         if (wt[j] <= i):
            table[i] = max(table[i], table[i - wt[j]] + val[j])
   print(table)

   return table[W]
```

```
[0, 0, 3, 3, 6, 10, 10, 13, 13, 16]
16
```