CS3383 - Assignment 4

Author: Shahriar Kariman

Due: Nov 18th

Question 1 - N-th Catalan Number

$$n = 0 \to (a) \to C(0) = 1$$
$$n = 1 \to (a)(b), (ab) \to C(1) = 2$$
$$n = 2 \to (a)(b)(c), (ab)(c), (a)(bc), (abc) \to C(2) = 4$$

If I have an expression of length n and I start from the 1_{st} character and keep expanding the size of the problem then i will keep reusing the calculated values.

```
def catalan(n):
  catalans = [0]*(n+1) # making an array of size n with all elements equal to 1
  catalans[0] = 1
  for i in range(1,n+1):
    for j in range(1,i+1):
      catalans[j] = catalans[j-1]*catalns[i-j]
  return catalans[j]
```

Question 2 - Smallest Subset Sum

So if I was to implement this recursively using brute force calculation I would do it like this:

```
smallest_subset = [0]*n

def smallestSubsetSum(S, g):
  if g<0: # Failed
    return {"success": False, "subset": []}
  elif g==0: # success
    s = []
    for i in range(len(S)):
      if S[i]<0:
        s.append(S[i])
    return {"success": False, "subset": s}
  # Otherwise
  for i in range(len(S)):
    k = S[i]
    if k>0:
      S[i] = -k
      result = smallestSubsetSum(S, g-k)
      if result["success"]:
        if len(result["subset"])<len(smallest_subset):
          smallest_subset = result["subset"]
    S[i] = k
    return {"success": len(smallest_subset)>0 and len(smallest_subset)<n, "subset": smallest_subset}
```

Comment: I know this isn't part of the solution I just thought it might help me solve the problem and it sort of did.

Actual Solution

If I use an $(n+1) \times (g+1)$ table with rows i and columns j where i is the number of elements and j is the value to sum upto and store the results whether or not a set can sum up to j would be whether the previous set could have summed up to j or if the subtracting the i_{th} element can get me a number the previous sets could have summed up to.

Note: The values inside the table are the size of the subset that would sum upto the number.

```
def dynamic_smallestSubSet(S, g):
  n = len(S)
  dp = [[float('inf')]*(g+1) for _ in range(n+1)]
  dp[0][0] = 0
  for i in range(1, n+1):
    for j in range(g+1):
      dp[i][j] = dp[i-1][j] # if I dont include the elemnt at i-1
      if j>=S[i-1]:
        dp[i][j] = min(dp[i-1][j],dp[i-1][j-S[i-1]]+1)
  if dp[n][g] == float('inf'):
    return None
  subset = []
  i, j = n, g
  while i>0 and j>0:
    if dp[i][j] != dp[i-1][j]: # i-1 was picked
      subset.append(S[i-1])
      j -= S[i-1]
    i -= 1
  return subset
```

Question 3 - Maximum Sum of any Contiguous Subarray

If a subarray already has a negative sum then there is no point of researching that sub array further. If I have start and end of the subarray stored somewhere and keep iterating through the loop updating the maximum value and start and end.

```
def maximumSubArraySum(A):
  max = current_sum = start = end = 0
  good_subarrays = []
  for i in range(len(A)):
    current_sum += A[i]
    if current_sum<0 and end>=start:
      good_subarrays.append({'start':start, 'end': end, 'sum': max})
      max, current_sum = 0
      start = end = i+1
    elif max<current_sum:
      end = i
      max = current_sum
  # After storing all of the subarrays with a positive vlaue that cant improve anymore
  # by adding new elements
  max = 0
  for i in range(len(good_subarrays)):
    sub_arr = good_subarrays[i]
    if sub_arr['sum'] > max:
      start = sub_arr['start']
      end = sub_arr['end']
  return A[start:end+1]
```

Final Solution

Well after all of that I realize the question wasn't asking for the subarray just the sum but since I spent so much time writing that I am not going to remove it.

But here is the shorter version:

```
def maximumSubArraySum(A):
  max = current_sum = 0
  for i in range(len(A)):
    current_sum += A[i]
    if current_sum<0:
      current_sum = 0
    elif max<current_sum:
      max = current_sum
  return max
```

So evidently we are looping through the entire array once so the time complexity if $\Theta(n)$.

Note: I agree this algorithm may not look like a dynamic programming algorithm but just because an algorithm does not have a big table doesn't mean it isn't a dynamic programming algorithm. In this case since I am storing the previous max and reusing it to get to the correct solution it technically qualifies as a dynamic programming algorithm.

Question 4 - Number Solitaire

So if I start from the $i = 0$ and $j = n$ and go inward my answer will always depend on from what end we pick the numbers:

$$ans(i, j) = max\ of \rightarrow \begin{cases} ans(i+1, j-1) + A_i \times A_j \\ ans(i+2, j) + A_i \times A_{i+1} \\ ans(i, j-2) + A_j \times A_{j-1} \end{cases}$$

Considering that the algorithm should be fairly straight forward. I put i at the end and kept growing the size of the problem much like question 1.

```
def numberSolitaire(A):
  n = len(A)
  dp = [[0]*(n) for _ in range(n)]
  for i in range(n-1, -1, -1): # n-1 to 0
    for j in range(i+1, n):
      val_1 = val_2 = val_3 = 0
      val_1 = dp[i+1][j-1] + A[i]*A[j]
      if i+1 < n:
        val_2 = dp[i+2][j] + A[i]*A[i+1]
      if j-1 > 0:
        val_2 = dp[i][j-2] + A[j]*A[j-1]
      dp[i][j] = max(val_1, val_2, val_3)
  return dp[0][n-1]
```

Question 5 - Longest Descending Subsequence

So at first I thought the question was asking for the longest consecutive subsequence at which case this *would have been* the correct algorithm:

```
def longetsDescendingSubsequence(A):
  lengths = [1]*len(A)
  length = 0
  for i in range(len(A)):
    if A[i]>A[i-1]:
      length = 0
    else:
      length += 1
    lengths[i] = length
  return max(lengths)
```

But then I read the question again and noticed that you can drop any element that doesn't fit in the order.

The answer is still pretty easy I just have to keep growing the size of the problem and store the result of each subproblem.

I chose to start from the left because I know that if element j comes after element i and $A[j] < A[i]$ then the longest sequence is going to be $Max\{result[j], result[i] + 1\}$ and if I keep calcualting the last elements result beased on the elements before it I will have my answer.

```
def longetsDescendingSubsequence(A):
  n = len(A)
  results = [1]*n
  for i in range(n-2,-1,-1): # n-2 to 0
    for j in range(i+1,n):
      if A[j]<A[i]:
        results[i] = max(results[i],results[j]+1)
  return max(results)
```

Question 6 - Distinct Subsequences Problem

Well I dont want to right a brute force algorithm and configure it to a dynamic algorithm. so I will just start by making a table and set the first row to all 1 s cause if T is empty then there should always be 1 subsequence in any S.

And then I figured as I am looping through every combination of i and j I could just include or not include the matching characters and keep going and if they didnt match then the count would be the same as if the i wasnt there.

Exhibit A

	j=0	r	a	b	b	i	t
i=0	0	0	0	0	0	0	0
r	1	1	0	0	0	0	0
a	1	1	1	0	0	0	0
b	1	1	1	1	0	0	0
b	1	1	1	2	1	0	0
b	1	1	1	3	3	0	0
i	1	1	1	3	3	3	0
t	1	1	1	3	3	3	3

Algorithm

```
def numDistinctSubsequence(S, T):
  m, n = len(S), len(T)
  # an (m+1) by (n+1) table
```

```python
dp = [[0]*(n+1) for _ in range(m+1)]

for i in range(m+1):
  dp[i][0] = 1

for i in range(1, m+1):
  for j in range(1, n+1):
    if S[i-1]==T[j-1]:
      dp[i][j] = dp[i-1][j-1] + dp[i-1][j]
    else:
      dp[i][j] = dp[i-1][j]
```

And the time complexity of the algorithm is obviously $\Theta(m \times n)$ since the first loop is executed m times and the inner loop executes $n \times m$ times as a result.