CS3383 - Assignment 4

Author: Shahriar Kariman

Due: Nov 18th

Question 1 - N-th Catalan Number

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

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 calcualted 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 calcualtion I would do it like this:

```
smallest_subset = [0]*n

def smallestSubsetSum(S, g):
  if g<0: # Failed
    return {"success": True, "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}
  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}
```

Now if S was in descending order I would essintially be doing a greedy search algorithm if I ran the smallestSubsetSum algorithm on a subset on the left and grow the subset each time.

```
def dynamic_smallestSubSet(S, g):
  mergeSort(S)
  n = len(S)
  table = [[float('inf')]*(g+1) for _ in range(n)]
```

```
  table[0][0] = 0
  for i in range(1, n+1):
    for j in range(g+1):
      pass
```

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]
```

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 programing algorithm but just because an algorithm does not have a big table doesn't mean it isn't a dynamic programing 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 programing 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 grwoing 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):
    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]
```

And cosidering the number of time the code in the inner loop is executed is:

$$n + (n-1) + ... \ + 1 = \sum_{k=0}^{n} n - k = \frac{n \times (n-1)}{2}$$

The run time analysis of the algorithm is $\Theta(n^2)$.

Question 5 - Longest Descending Subsequence

This one is easy I can just store the value of the longest descending subsequence for every element upto that element and store the value for each element inside an array of the same size and I can loop through it and get the biggest number.

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

Offical Question for the Prof (if he ever reads this)

Why are the questions not in order of difficaulty? This should have been the very first question.

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
  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.