ADA Assignment 4

Ankur Sharma

March 2018

Partners: Anvit Mangal (2016135) and Ishaan Bassi (2016238).

Problem 1

Given n, how many structurally unique BST's (binary search trees) that store values 1...n? For e.x if n=3 we have 5 unique BST. Design a DP solution to this problem.

Solution:
From inspection, we can see that the given problem has sub-problems. For e.x. let total no. of nodes = n. we can have n nodes as a root. If we take i^{th} node as a root, no. of nodes on the left side of the root = i - 1 and no. of nodes on the right side of the root = n - i. Let no. of BST for n nodes = BST(n). Now, **No. of BST when i^{th} node is root = BST(i-1) * BST(n-i)**. If we fix '1' as a root, no. of BST = BST(0) * BST(n-1). As mentioned above, we have n options [1,2,3,...,n] to fix root. Now, if we sum this up, we get BST(n) = BST(1-1) * BST(n-1) + BST(2-1) * BST(n-2) + ... + BST(n-1) * BST(n-n). or, BST(n) = BST(0) * BST(n-1) + BST(1) * BST(n-2) + ... + BST(n-1) * BST(0).
Let n=2, BST(2) = BST(0)*BST(1) + BST(1) * BST(0). It can be seen that there are **overlapping sub-problems** which can be handled using DP.
Optimal substructure: We are using i-1 and n-i to calculate BST when i^{th} element is fixed as a root.
DP Solution Pseudo-code:
dp: 1-D array of n + 1 size, all values filled with -1 initially.
Procedure BST(n, dp):

```
if(n==0 or n==1): # base case
    return 1;
result = 0;
for i in range(1,n): # fixing nodes from 1 to n (both inclusive)
    # if not calculated then compute
    dp[i-1] = (dp[i-1] == -1) ? BST(i-1, dp) : dp[i-1]; # ternary operator
```

```
    # if not calculated then compute
    dp[n-i] = (dp[n-i] == -1) ? BST(n-i, dp) : dp[n-i];
    result = result + (dp[i-1] * dp[n-i]);
  return result;
```

Time Complexity: O(n^2)

Problem 2

As Candidates Chess Tournament is going one of your friend has come with an interesting problem and need your help to devise a DP solution (top-down as well bottom up) for the given problem. On an NxN (N range 0-25) chessboard, a knight starts at the r-th row and c-th column and attempts to make exactly K(K range 0-100) moves. The rows and columns are 0 indexed, so the top-left square is (0, 0), and the bottom-right square is (N-1, N-1). A chess knight has 8 possible moves it can make. Each time the knight is to move, it chooses one of eight possible moves uniformly at random (even if the piece would go off the chessboard) and moves there.The knight continues moving until it has made exactly K moves or has moved off the chessboard. Return the probability that the knight remains on the board after it has stopped moving. The knight always initially starts on the board.Explain the time complexity of your solution. Sample example: Input: N=3, k=1, r=0, c=0 Output: 0.25 Explanation: There are two moves (to (1,2), (2,1)) that will keep the knight on the board.The total probability the knight stays on the board is 0.25.

Solution:
We need to find the probability of a knight to remain in chessboard after it has moved exactly K steps starting from given (x,y) which denotes initial coordinates of knight on the chessboard. At each step, knight can choose from 8 positions (some of them may lie outside and some of them may lie inside). Now, Probability of reaching (x,y) on a chessboard can be found if we know the probability of reaching neighbors(x,y) in k-1 steps hence **optimal substructure**. Also, there can be multiple **overlapping sub-problems** as knight can reach a cell on chessboard from multiple cells using multiple steps. Therefore, it is a recursion problem with overlapping sub-problems.
Let us take 3-D array dp[n][n][k] s.t. dp[i][j][u] stores the probability of knight to remain on chessboard starting from (i,j) and taking u steps.
Bottom-up Approach:
Base-Case: When K=0, Probability of knight to remain on chessboard = 1 as knight is already on the chessboard.
Pseudo-Code:

```
Procedure func(a, b, n, K):
  for (i,j) in chessboard: # for all cells in chessboard
    dp[i][j][0] = 1
  for u in range(1, K): # loop from 1 to K
    for i in range(0, n): # loop from 0 to n-1
```

```
        for j in range(0,n): # loop from 0 to n-1
            for (x,y) in neighbors(i,j) : # cells knight can reach from (x,y)
                temp=0
                if (x,y) is inside chessboard:
                    temp = dp[x][y][k-1] + temp
                dp[i][j][k] = temp / 8
    return dp[a][b][K]
```

Top-down Approach:

Base-Case:

If knight is outside the chessboard, it cannot enter the chessboard again (given), therefore, probability = 0

If knight is inside the chessboard and knight has no further steps to take then knight stays inside it and probability becomes 1.

Pseudo-Code:

```
Procedure func(a, b, n, K):
    if cell(a,b) outside chessboard:
        return 0
    if K == 0 :
        return 1
    temp = 0
    for (x,y) in neighbors(a,b): # cells knight can reach from (x,y)
        if (x,y) is inside chessboard
            if dp[x][y][K-1] == -1 : # not calculated yet
                dp[x][y][K-1] = func(x, y, n, K-1)
            else: total = dp[x][y][K-1] + total
    total = total / 8
    return total
```

Time-Complexity:

In bottom-up approach, we have 4 nested for-loops hence, O(K * n * n * 8) becomes O(K * n^2).

In top-down approach, we calculating dp array for all (may be 8)neighbors of given cell with K steps but base-case is at K=0 and as a knight can at most go to n^2 cells inside chessboard, therefore, O(K * n^2).

Problem 3

Given a sequence of n real numbers, a1, a2, ..., an, give an algorithm for finding a contiguous sub-sequence for which the value of the sum of the elements is maximized.

Solution:

Let 'A' be the given array. We need to find a contiguous sub-array such that the sum of the elements in the sub-array is maximized. A **Greedy Approach** is used here. There are two cases to be considered:

Case I: All the elements in A are negative - In this case, we simply return an

index with max value.
Case II: There is at-least one positive element in A - The idea is to keep track of 'maximum positive contiguous sum' and the first and last indexes of sub-array. We take 'temp' variable to calculate the sum so far and update 'maximum positive contiguous sum' by temp if temp is greater than 'maximum positive contiguous sum' and thereby updating 'first' and 'last' variables which store first and last index of maximum sum subarray respectively.
Pseudo-Code:

```
Procedure findMaxSubArray(A):
   n = A.size()
   # check if all elements in A are negative
   case = 1
   minIndex = 0
   for i in range(0, n): # loop from 0 to n-1
      if A[i] > A[minIndex] :
         minIndex = i
      if A[i] ≥ 0 :
         case = 2
         break;
   if case == 1 :
      print A[minIndex]
      return;
   # 'maximum positive contiguous sum' = mpcs
   mpcs = temp = first = last = tempi = 0
   for i in range(0, n): # loop from 0 to n-1
      temp = A[i] + temp
      if mpcs < temp :
         mpcs = temp
         first = tempi
         last = i
   if temp < 0 :
      tempi = i + 1
      temp = 0
   p = first
   while ( p ≤ last ):
      print A[p++]
   return;
```

Time Complexity: O(n)

Problem 4

Suppose you are given an array of n integers a1, ..., an between 0 and M. Give an algorithm for dividing these integers into two sets x and y such that the difference of the sum of the integers in each set, is minimized.

Solution:
Let S be an array of all given integers and we divide it in S1 and S2 s.t. mod of difference of the sum of the integers in each array is minimized. Another way to look at the problem is we take out elements from S and put them in S1 and rename S2 as S. Therefore, $\forall$ elements in S, we either include it in S1 or include it in S2(or S).
As both elements in S are integers and their sum is an integer, we can memorize the sub-problems using array. Let, initially, sum of all elements in S is denoted by 'T'. Let us take a 2-D array dp[n+1][T+1] s.t. dp[i][j] = 1 if $\exists$ a subset in S from index 0 to i-1 (both inclusive) s.t it has sum equal to 'j'. Else 0.
We will try to find the j such that dp[n][j] = 1 and | j - (T - j) | is minimum, where 'j' is the sum of elements in S1 and (T - j) is sum of elements in S2. We will do this in bottom-up approach.
As it can be seen from Pseudo-code that to calculate dp[i][j], we are either including i^{th} elements in S1 or not and it is further calculated using dp[i-1][j], dp[i-1][j - S[i]] and, hence **optimal substructure and overlapping subproblems**.
Base-Case: when j = 0, $\forall$ i dp[i][j] = 1 because $\forall$ i $\exists$ empty subset (no elements) implies sum = 0. When i = 0 (empty set) and j>0, dp[i][j] = 0 because there are no elements to make sum>0.
Pseudo-Code:

```
Procedure divide(S):
   # initialize dp as described in Base-Case.
   for i in range(0, n): # loop from 0 to n
      dp[i][0] = 1
   for j in range(1, T): # loop from 1 to T
      dp[0][j] = 0

   for i in range(1, n): # loop from 1 to n
      for j in range(1, T): # loop from 1 to T
         #either we include i^th element in S1 or we don't.
         if (j - S[i] ≥ 0)
            dp[i][j] = max(dp[i-1][j], dp[i-1][j - S[i]])
         else: dp[i][j] = dp[i-1][j]

   # find j s.t. | dp[n][j] - (T - dp[n][j]) | is minimum.
   minimumDiff = Infinity
   for j in range(0, T): # loop from 0 to T
      temp = abs( dp[n][j] - (T - dp[n][j]))
      if( temp < minimumDiff ):
         minimumDiff = temp
   return minimumDiff;
```

Time Complexity: O(n * T)