problem 1 *Easier DP*

Consider a Galton board (see one at https://en.wikipedia.org/wiki/Galton_board). If you are unfamiliar with a Galton board, read that article. In that board, each grid spot has either a peg or is empty.

The question is, how many possible paths are there in such a board from a given starting location to a given ending location?

You are given a two-dimensional array M, of dimensions $w \times h$, which contains whether or not there is a peg. For example, the Galton board shown in the Wikipedia article is of size 23×10, and would be represented as:

$$M = \begin{matrix}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & \mathbf{1} & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1
\end{matrix}$$

We define the lower-left corner as $(0,0)$ (aka `[0][0]`). You are given a starting location (s_x, s_y), which will be on the top row. In this example, the starting location is $(11, 9)$, since we are indexing from 0 – the 1 in this cell is bolded in the array above. You are given an ending location (e_x, e_y). The ending location will always be in the bottom row, so $e_y = 0$.

Given such a setup, the algorithm is to determine how many possible paths there are from (s_x, s_y) to (e_x, e_y).

For this problem, we will be following the four steps of developing a dynamic programming solution as presented in lecture.

1. Step 1: *Recognize what the sub-problems are.* Explain what the sub-problems are for this problem. This can be in pseudo-code, as explained below, or as an English explanation.

   Solution: This problem, and some of the code below, is from a similar problem at https://www.geeksforgeeks.org/dsa/unique-paths-in-a-grid-with-obstacles/.

   For each given open spot (a 0 in the array), the ball can come from either of the two spots above, so it is just the number above and to the right plus the number above and to the left.

2. Step 2: *Identify the recursive structure of the problem in terms of its sub-problems (At the top level, what is the "last thing" done?).* Show the recursive structure – this is likely done by showing the recursive calls. This can be in pseudo-code, as explained below, or as an English explanation.

   Solution: $combos(x,y) = combos(x-1,y+1) + combos(x+1,y+1)$

3. Step 3: *Formulate a data structure (array, table) that can look-up solution to any sub-problem in constant time.* Explain the array size and what it holds.

   Solution: A 2-d array that matches the table size above. Fill the top row with all 0's. Fill each non-empty cell (a 1 in the grid above) with a -1.

4. Step 4a: *Develop an algorithm that loops through data structure solving each sub-problem one at a time.* This should be a **top-down (i.e., recursive)** algorithm.

   You should write it in Python-like pseudo-code – an example of how to write pseudo-code in LaTeX is shown below. Note that the `lstlisting` environment cannot be inside the `\solution{}` block, and must be outside it.

   Solution:

```
M = ...
C = new int[M.wdith][M.height]

def combos(sx,sy,ex,ey):
  for x in range(M.width):
    for y in range(M.height):
      C[x][y] = -1
  for x in range(M.width):
    C[x][M.height-1] = 0
  C[sx][sy] = 1
  combosRec(sx,sy)
  return C[ex][ey]

def combosRec(x,y):
  if x < 0 or y < 0 or x >= M.width or y >= M.height:
    return 0
  if C[x][y] != -1:
    return C[x][y]
  result = combos(x-1,y-1) + combos(x+1,y-1)
  C[x][y] = result
  return result

def fib(n):
  if n <= 1:
    return 1
  else
    return fib(n-1) + fib(n-2)
```

5. Step 4b: *Develop an algorithm that loops through data structure solving each sub-problem one at a time.* This should be a **bottom-up (i.e., iterative)** algorithm. You should write it in pseudo-code.

   Solution:

```
M = ...
C = new int[M.wdith][M.height]

def combos(sx,sy,ex,ey):
  for x in range(M.width):
    for y in range(M.height):
      C[x][y] = -1
  C[sx][sy] = 1
  for y in range(M.height-2,-1,-1):
    for x in range(M.width):
      if x > 0:
        C[x][y] += C[x-1][y+1]
      if x < M.width:
        C[x][y] += C[x+1][y+1]
  return C[ex][ey]
```

PROBLEM 2 *Airplane loading*

When loading aircraft, one has to ensure that the weight on the left side of the plane is roughly the same as the weight on the right side of the aircraft. This is true for both passengers and cargo. Consider the problem of loading cargo pallets into an aircraft. Each cargo pallet has a *weight*, which we will represent as a positive integer. Your task is to see if the pallets can be split into two parts of equal weight – one for the left side of the aircraft and one for the right side. (For this problem, we will ignore the finite capacity of how many actual pallets can fit inside an airplane).

Given a series of positive integer pallet weights $w_1, w_2, \ldots, w_n$, check if you can split them into two parts such that the weight of each is the same.

As an example, pallets of weights $5, 10, 2, 1, 2$ can be split into two parts, both of weight 10: $1, 2, 2, 5$ and 10. Conversely, pallets of weights $2, 7, 4$ cannot be split into two parts of equal weight.

For this problem, we will be following the four steps of developing a dynamic programming solution as presented in lecture.

1. Step 1: *Recognize what the sub-problems are.* Explain what the sub-problems are for this problem. This can be in pseudo-code, as explained below, or as an English explanation.

   Solution: First, calculate the sum of the array. If it's odd, there is no way to split the weights into two parts (all weights, and results, are integer). If it is even, calculate $sum/2$ – the goal is to find a subset that sums to $sum/2$ (which means the rest of the set will also sum to $sum/2$).

   When given an array to sum, either the last element is in the array or it is not. So we call ourselves recursively in both cases. This will lead to a $\Theta(2^n)$ implementation via recursion.

2. Step 2: *Identify the recursive structure of the problem in terms of its sub-problems (At the top level, what is the "last thing" done?).* Show the recursive structure – this is likely done by showing the recursive calls. This can be in pseudo-code, as explained below, or as an English explanation.

   Solution: Our function is going to be `isSubsetSum(n, array, sum)`. We are going to consider the elements from the last back to the first. The parameter `n` is how many are left in the array to consider. `sum` is the sum we are trying to achieve. The idea is that `isSubsetSum(n,array,sum)` will consider two recursive calls: with the last element (n) in the sum to reach sum, and without.

   Base cases: if $n = 0$ (there are no more elements left to consider) and $sum \neq 0$ (there is still a sum to achieve), then reutrn `false` (you can't have a positive sum with zero elements). The

other base case is that if $sum = 0$ then return `true` (don't include any more elements to sum to 0).

The code below was obtained from https://www.geeksforgeeks.org/dsa/partition-problem-dp-18/

```
def isSubsetSum(n, arr, sum):

    # base cases
    if sum == 0:
        return True
    if n == 0:
        return False

    # If element is greater than sum, then ignore it
    if arr[n-1] > sum:
        return isSubsetSum(n-1, arr, sum)

    # Check if sum can be obtained by any of the following
    # (a) including the current element
    # (b) excluding the current element
    return isSubsetSum(n-1, arr, sum) or \
         isSubsetSum(n-1, arr, sum - arr[n-1])

def equalPartition(arr):

    # Calculate sum of the elements in array
    arrSum = sum(arr)

    # If sum is odd, there cannot be two
    # subsets with equal sum
    if arrSum % 2 != 0:
        return False

    # Find if there is subset with sum equal
    # to half of total sum
    return isSubsetSum(len(arr), arr, arrSum // 2)

if __name__ == "__main__":
    arr = [1, 5, 11, 5]
    if equalPartition(arr):
        print("True")
    else:
        print("False")
```

3. Step 3: *Formulate a data structure (array, table) that can look-up solution to any sub-problem in constant time.* Explain the array size and what it holds.

   Solution: A 2-D array that holds `true` or `false` whether the first n elements can sum to sum.

4. Step 4a: *Develop an algorithm that loops through data structure solving each sub-problem one at a time.* This should be a **top-down (i.e., recursive)** algorithm.

You should write it in Python-like pseudo-code – an example of how to write pseudo-code in LaTeX is shown below. Note that the `lstlisting` environment cannot be inside the `\solution{}` block, and must be outside it.

Solution: The code below was obtained from https://www.geeksforgeeks.org/dsa/partition-problem-dp-18/

```
# Python program to partition a Set
# into Two Subsets of Equal Sum
# using top down approach
def isSubsetSum(n, arr, sum, memo):

    # base cases
    if sum == 0:
        return True
    if n == 0:
        return False

    if memo[n-1][sum] != -1:
        return memo[n-1][sum]

    # If element is greater than sum, then ignore it
    if arr[n-1] > sum:
        return isSubsetSum(n-1, arr, sum, memo)

    # Check if sum can be obtained by any of the following
    # (a) including the current element
    # (b) excluding the current element
    memo[n-1][sum] = isSubsetSum(n-1, arr, sum, memo) or\
            isSubsetSum(n-1, arr, sum - arr[n-1], memo)
    return memo[n-1][sum]

def equalPartition(arr):

    # Calculate sum of the elements in array
    arrSum = sum(arr)

    # If sum is odd, there cannot be two
    # subsets with equal sum
    if arrSum % 2 != 0:
        return False

    memo = [[-1 for _ in range(arrSum+1)] for _ in range(len(arr))]

    # Find if there is subset with sum equal
    # to half of total sum
    return isSubsetSum(len(arr), arr, arrSum // 2, memo)

if __name__ == "__main__":
    arr = [1, 5, 11, 5]
    if equalPartition(arr):
        print("True")
    else:
```

```
        print("False")
```

5. Step 4b: *Develop an algorithm that loops through data structure solving each sub-problem one at a time.* This should be a **bottom-up (i.e., iterative)** algorithm. You should write it in pseudo-code.

   Solution: The code below was obtained from https://www.geeksforgeeks.org/dsa/partition-problem-dp-18/

```
# Python program to partition a Set
# into Two Subsets of Equal Sum
# using top up approach
def equalPartition(arr):

    # Calculate sum of the elements in array
    arrSum = sum(arr)
    n = len(arr)

    # If sum is odd, there cannot be two
    # subsets with equal sum
    if arrSum % 2 != 0:
        return False

    arrSum = arrSum // 2

    # Create a 2D array for storing results
    # of subproblems
    dp = [[False] * (arrSum + 1) for _ in range(n + 1)]

    # If sum is 0, then answer is true (empty subset)
    for i in range(n + 1):
        dp[i][0] = True

    # Fill the dp table in bottom-up manner
    for i in range(1, n + 1):
        for j in range(1, arrSum + 1):
            if j < arr[i - 1]:

                # Exclude the current element
                dp[i][j] = dp[i - 1][j]
            else:

                # Include or exclude
                dp[i][j] = dp[i - 1][j] or dp[i - 1][j - arr[i - 1]]

    return dp[n][arrSum]

if __name__ == "__main__":
    arr = [1, 5, 11, 5]
    if equalPartition(arr):
        print("True")
    else:
        print("False")
```