16 Apr 2022, 12 Nov 2022, 15 Apr 2023, 16 Nov 2019

Your task is to implement count_plays that counts the number of possible ways starting from a given square to the finish square without ever landing on top of any chute. The function takes as input a starting position (a number from 1 to N −1) and a representation of a chutes and-ladders game board. A player need not exactly land on the final square to finish a play. For example, if player is 2 spots away for the the final square and rolls a 6, it is counted as a play.The board is represented by a tuple (num_squares, ladders, chutes).

In [1]:

```
board1 = (14, {2: 12, 7: 12}, {13: 3, 9: 4})
board2 = (10, {5: 8}, {9: 1, 7: 4})
```

In [2]:

```
#without ever landing on top of any chute
#each function only runs one thing at a time hence need to use elif

def subsetSum(target, currentSum, array, countArray, board):


    if board[2].get(currentSum) != None:

        return

    elif currentSum >= target :

        countArray.append(1)
        return

    elif board[1].get(currentSum) != None:

        subsetSum(target, board[1].get(currentSum), array, countArray,board)

    else:

        for num in array:

            if currentSum < target :

                subsetSum(target, currentSum+num, array, countArray,board)
```

In [3]:

```
def num_plays(num, board):
    countArray = []
    array = [1,2,3,4,5,6]
    subsetSum(board[0], num, array, countArray, board)
    totalPaths = 0
    for i in range(len(countArray)):

        totalPaths += 1

    return totalPaths
```

In [4]:

```
num_plays(5, board2)
```

Out[4]:

```
5
```

In [5]:

```
num_plays(8, board2)
```

Out[5]:

```
5
```

In [6]:

```
num_plays(3, board2)
```

Out[6]:

```
37
```

In [7]:

```
num_plays(13, board1)
```

Out[7]:

```
0
```

In [8]:

```
num_plays(12, board1)
```

Out[8]:

```
5
```

An interval is a pair of integers in the form (a, b) where a < b. It represents an interval of integers from a to b inclusive. Two intervals can be merged to be represented by one interval if they overlap. The following helper function merge takes in two intervals as inputs, and returns a new merged interval if the two intervals overlap, or False otherwise. The implementation is given as follows.Using merge , your task is to implement the function merge_intervals that takes in an arbitrary number of intervals and returns a list with the minimum number of merged intervals that covers all the intervals in the input. You can return the list of intervals in any order.

In [9]:

```
def merge(a, b):
    if a[1] < b[0] or b[1] < a[0]:
        return False
    else:
        return (min(a[0], b[0]), max(a[1], b[1]))
```

In [10]:

```
def merge_list(intervals):

    initialLength = len(intervals)

    for i in range(initialLength):

        flag = False

        newInterval = intervals.pop()

        for j in range(len(intervals)):

            result = merge(newInterval, intervals[j])

            if result != False:
                flag = True
                intervals[j] = result
                break

        if flag == False:

            intervals.append(newInterval)


```

In [11]:

```
def merge_intervals(*intervals):

    interval_List = []

    for x in intervals:

        interval_List.append(x)

    merge_list(interval_List)

    return interval_List
```

In [12]:
```
merge_intervals((1, 2), (3, 4))
```

Out[12]:
```
[(1, 2), (3, 4)]
```

In [13]:
```
merge_intervals((1, 2), (2, 4))
```

Out[13]:
```
[(1, 4)]
```

In [14]:
```
merge_intervals((1, 2), (3, 4), (2, 6))
```

Out[14]:
```
[(1, 6)]
```

In [15]:
```
merge_intervals((7, 12), (3, 9), (1, 4))
```

Out[15]:
```
[(1, 12)]
```

In [16]:
```
merge_intervals((7, 9), (3, 5), (10, 13), (1, 15))
```

Out[16]:
```
[(1, 15)]
```

Implement the function flatten_dictionary that takes in a dictionary dict, and separator value sep and returns a flattened dictionary if dict is nested. In case of duplicate keys generated after flattening, consider the first value assigned to the key, see the last sample in the example.

In [17]:
```
x = {}
str(type(x))
```

Out[17]:
```
"<class 'dict'>"
```

In [18]:
```
def flatten_helper(elt,sep,currentkey,final):

    if str(type(elt)) == "<class 'dict'>":

        for key in elt.keys():

            flatten_helper(elt.get(key), sep, currentkey + sep + str(key), final)

    else:

        if final.get(str(currentkey)) == None:

            final[str(currentkey)] = elt

        return
```

In [19]:
```
def flatten_dictionary(dictionary,sep):

    final = {}
    for key in dictionary.keys():

        flatten_helper(dictionary.get(key), sep, str(key), final)

    return final
```

In [20]:
```
nested_dict={'a': 1, 'b': {'c': 2, 'd': {'e': 3}}}
flatten_dictionary(nested_dict, '_')
```

Out[20]:
```
{'a': 1, 'b_c': 2, 'b_d_e': 3}
```

In [21]:
```
nested_dict={'C': {'S': {'1': {'0': {'1': {'0':'S'}}}}}}
flatten_dictionary(nested_dict, '')
```

Out[21]:
```
{'CS1010': 'S'}
```

In [22]:
```
nested_dict={2: 1, 4: {6: 2, 8: {10: 3}, 12:4}, 14:5}
flatten_dictionary(nested_dict, '-')
```

Out[22]:
```
{'2': 1, '4-6': 2, '4-8-10': 3, '4-12': 4, '14': 5}
```

In [23]:
```
nested_dict={(1,2): { 3: ['a','b', 'c'], (4,5): {(6,7):
['d','e,','f','g'] }}}
flatten_dictionary(nested_dict, '*')
```

Out[23]:
```
{'(1, 2)*3': ['a', 'b', 'c'], '(1, 2)*(4, 5)*(6, 7)': ['d', 'e,', 'f', 'g']}
```

In [24]:
```
nested_dict={'b_c': 5, 'b': {'c': 3}}
flatten_dictionary(nested_dict, '_')
```

Out[24]:
```
{'b_c': 5}
```

A game of matchsticks is played by first arranging matchsticks in a sequence. Players then take turns to remove any number of consecutive matchsticks from the game. When a matchstick is removed, it creates a gap, splitting the matchsticks into separate piles.

Implement a function valid_move(game, start, end) that takes in a game state in the above-mentioned representation, and the starting and ending indexes of matchstick to remove. You may assume that end will not be smaller than start .

In [25]:
```
def valid_move_index(game, start, end):

    length = len(game)

    if length == 1 and start <= game[0][-1] and end >= game[0][0]:

        return 0


    for i in range(0,length):


        if i == 0:

            if start >= game[i][0] and start <= game[i][-1] and end < game[i+1][0]

                return i

        elif i == length -1 :
            if start > game[i-1][-1] and start <= game[i][-1] and end <= game[i][-1

                return i

        else:

            if start > game[i-1][-1] and start <= game[i][-1] and end < game[i+1][

                return i


    return -1
```

In [26]:
```
def valid_move(game, start, end):

    if valid_move_index(game, start, end) < 0:

        return False

    else:

        return True
```

In [27]:
```
game = [[1, 3], [8, 9], [14,20]]
valid_move(game, 2, 8)
```

Out[27]:
```
False
```

In [28]:
```
valid_move(game, 4, 7)
```

Out[28]:
```
False
```

In [29]:
```
valid_move(game, 2, 6)
```

Out[29]:
```
True
```

Implement the function make_move(game, start, end) that takes in a game state, and the starting and ending indexes of matchstick to remove. The function updates and returns the game state with the specified matchsticks removed.

In [30]:
```
def make_move(game, start, end):

    length = len(game)
    index = valid_move_index(game, start, end)
    if index < 0:

        return game

    if start <= game[index][0] and end >= game[index][-1]:

        game.pop(index)
        return game

    elif start <= game[index][0] and end < game[index][-1] :

        game[index][0] = end + 1
        return game

    elif start > game[index][0] and end >= game[index][-1]:
        game[index][-1] = start - 1
        return game

    elif start > game[index][0] and end < game[index][-1]:

        newArray = [end + 1, game[index][-1]]
        game[index][-1] = start - 1
        game.insert(index+1,newArray)
        return game
```

In [31]:

```
game = [[1, 20]]
make_move(game, 10, 13)
```

Out[31]:

```
[[1, 9], [14, 20]]
```

In [32]:

```
make_move(game, 4, 7)
```

Out[32]:

```
[[1, 3], [8, 9], [14, 20]]
```

In [33]:

```
make_move(game, 15, 17)
```

Out[33]:

```
[[1, 3], [8, 9], [14, 14], [18, 20]]
```