Notebook

August 2, 2024

```
[ ]: #### https://leetcode.com/problems/sudoku-solver/description/
```

```
[ ]: #### https://leetcode.com/problems/unique-paths-iii/description/
     #### 1 represents the starting square. 2 represents the ending square. 0␣
      ↪represents the empty square. -1 represents an obstacle.

     from typing import List

     class Solution:
         def uniquePathsIII(self, grid: List[List[int]]) -> int:
             m, n = len(grid), len(grid[0])

             # iterate through the grid to get relevant info
             start = None  # to store starting point
             count = 0  # to count number of squares to walk over
             for i in range(m):
                 for j in range(n):
                     count += grid[i][j] == 0
                     if not start and grid[i][j] == 1:
                         start = (i, j)

             def backtrack(i: int, j: int) -> int:
                 nonlocal count
                 result = 0
                 for x, y in ((i-1, j), (i+1, j), (i, j-1), (i, j+1)):
                     # border check
                     if 0 <= x < m and 0 <= y < n:
                         if grid[x][y] == 0:
                             # traverse down this path
                             grid[x][y] = -1
                             count -= 1
                             result += backtrack(x, y)
                             # backtrack and reset
                             grid[x][y] = 0
                             count += 1
                         elif grid[x][y] == 2:
                             # check if all squares have been walked over
```

```
                        result += count == 0
            return result

        # perform DFS + backtracking to find valid paths
        return backtrack(start[0], start[1])
```

[ ]:
```
# https://leetcode.com/problems/expression-add-operators/description/
# Input: num = "123", target = 6   Output: ["1*2*3","1+2+3"]

class Solution:
    def addOperators(self, num: str, target: int) -> List[str]:
        ans = []
        self.backtrack(num, target, 0, 0, "", 0, ans)
        return ans

    def backtrack(self, num: str, target: int, current: int, index: int, temp:␣
 ↪str, prev: int, ans: List[str]):
        if index == len(num) and current == target:
            ans.append(temp)
            return

        if index == len(num):
            return

        cnum = 0
        snum = ""

        for i in range(index, len(num)):
            if i > index and num[index] == '0':
                break

            cnum = cnum * 10 + int(num[i])
            snum += num[i]

            if index == 0:
                self.backtrack(num, target, cnum, i + 1, snum, cnum, ans)
            else:
                self.backtrack(num, target, current + cnum, i + 1, temp + '+' +␣
 ↪snum, cnum, ans)
                self.backtrack(num, target, current - cnum, i + 1, temp + '-' +␣
 ↪snum, -cnum, ans)
                self.backtrack(num, target, current - prev + prev * cnum, i +␣
 ↪1, temp + '*' + snum, prev * cnum, ans)
```