Notebook

August 6, 2024

[ ]:

```
from typing import List
from functools import lru_cache
import math
```

[ ]:

```
#### https://leetcode.com/problems/longest-palindromic-subsequence/description/

class Solution:
    def longestPalindromeSubseq(self, s: str) -> int:
        n = len(s)

        @lru_cache(None)
        def dp(l, r):
            if l > r: return 0  # Return 0 since it's empty string
            if l == r: return 1  # Return 1 since it has 1 character
            if s[l] == s[r]:
                return dp(l+1, r-1) + 2
            return max(dp(l, r-1), dp(l+1, r))

        return dp(0, n-1)
```

[ ]:

```
#### https://leetcode.com/problems/
 ↪minimum-insertion-steps-to-make-a-string-palindrome/

"""
class Solution {
public:
    int dp[501][501];
    int recurr(string &s, int i, int j) {
        if (i >= j) return 0;
        if (dp[i][j] != -1) return dp[i][j];
        return dp[i][j] = (s[i] == s[j]) ? recurr(s, i+1, j-1) : 1 +␣
 ↪min(recurr(s, i+1, j), recurr(s, i, j-1));
    }
    int minInsertions(string s) {
        int n = s.size();
        memset(dp, -1, sizeof(dp));
        return recurr(s, 0, n-1);
    }
};
"""
```

[ ]:

```
#### https://leetcode.com/problems/longest-common-subsequence/description/

class Solution:
    def longestCommonSubsequence(self, s1: str, s2: str) -> int:
        m = len(s1)
        n = len(s2)
        memo = [[-1 for _ in range(n + 1)] for _ in range(m + 1)]
        return self.helper(s1, s2, 0, 0, memo)

    def helper(self, s1, s2, i, j, memo):
        if memo[i][j] < 0:
            if i == len(s1) or j == len(s2):
                memo[i][j] = 0
            elif s1[i] == s2[j]:
                memo[i][j] = 1 + self.helper(s1, s2, i + 1, j + 1, memo)
            else:
                memo[i][j] = max(
                    self.helper(s1, s2, i + 1, j, memo),
                    self.helper(s1, s2, i, j + 1, memo),
                )
        return memo[i][j]
```

[ ]:

```
#### longest alternating subsequence
#### a1 > a2 < a3 > a4 < a5 > a6 < a7

def longest_alternating_subsequence(arr):
    if len(arr) == 0:
        return 0

    up, down = 1, 1

    for i in range(1, len(arr)):
        if arr[i] > arr[i-1]:
            up = down + 1
        elif arr[i] < arr[i-1]:
            down = up + 1

    return max(up, down)

# Example usage
arr = [1, 5, 4]
print(longest_alternating_subsequence(arr))  # Output: 3
```

[ ]:

```
#### https://leetcode.com/problems/max-dot-product-of-two-subsequences/
 ↪description/
#### LCS pattern
```

[ ]:

```
#### https://leetcode.com/problems/shortest-common-supersequence/description/
```

[ ]:

```
#### https://leetcode.com/problems/unique-paths/description/

"""
Input: m = 3, n = 7
Output: 28
"""

class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        dp = [[0] * n for _ in range(m)]
        for r in range(m):
            for c in range(n):
                if r == 0 or c == 0:
                    dp[r][c] = 1
                else:
                    dp[r][c] = dp[r-1][c] + dp[r][c-1]
        return dp[m-1][n-1]


#### https://leetcode.com/problems/unique-paths-ii/description/
#### Follow up: Now consider if some obstacles are added to the grids. How many␣
 ↪unique paths would there be?

"""
Input: obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]
Output: 2
Explanation: There is one obstacle in the middle of the 3x3 grid above.
There are two ways to reach the bottom-right corner:
1. Right -> Right -> Down -> Down
2. Down -> Down -> Right -> Right
"""

class Solution:
    def uniquePaths2(self, obstacleGrid) -> int:
        m, n = len(obstacleGrid), len(obstacleGrid[0])
        dp = [[0] * n for _ in range(m)]
        if obstacleGrid[0][0] == 1:
            return 0
        for r in range(m):
            for c in range(n):
                if obstacleGrid[r][c] == 1:
                    dp[r][c] = 0
                else:
                    if r == 0 or c == 0:
                        dp[r][c] = 1
                    else:
                        dp[r][c] = dp[r-1][c] + dp[r][c-1]
        return dp[m-1][n-1]


#### https://leetcode.com/problems/unique-paths-iii/description/
#### 1 represents the starting square. 2 represents the ending square. 0 represents empty squares we can walk over. -1 represents obstacles.
### backtracking solution

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/dungeon-game/description/
#### return the minimum initial health you will need to make it through the␣
 ↪dungeon

#### Binaary Search + DP
"""
Input: dungeon = [[-2,-3,3],[-5,-10,1],[10,30,-5]]
Output: 7
Explanation: The initial health of the knight must be at least 7 if he follows␣
 ↪the optimal path: RIGHT-> RIGHT -> DOWN -> DOWN.
"""
class Solution:
    def calculateMinimumHP(self, grid: List[List[int]]) -> int:
        m, n = len(grid), len(grid[0])

        def isGood(initHealth):
            dp = [[0] * n for _ in range(m)]
            dp[0][0] = initHealth + grid[0][0]
            for r in range(m):
                for c in range(n):
                    if r > 0 and dp[r-1][c] > 0:
                        dp[r][c] = max(dp[r][c], dp[r-1][c] + grid[r][c])
                    if c > 0 and dp[r][c-1] > 0:
                        dp[r][c] = max(dp[r][c], dp[r][c-1] + grid[r][c])
            return dp[m-1][n-1] > 0

        left = 1
        right = 1000 * (m + n) + 1
        ans = right
        while left <= right:
            mid = left + (right - left) // 2
            if isGood(mid):
                ans = mid
                right = mid - 1
            else:
                left = mid + 1
        return ans

#### Top Down
class Solution:
    def recurr(self, dungeon, r, c, dp):
        rows, cols = len(dungeon), len(dungeon[0])

        if r == rows - 1 and c == cols - 1:
            if dungeon[r][c] <= 0:
                return abs(dungeon[r][c]) + 1
            return 1

        if dp[r][c] != float('inf'):
            return dp[r][c]

        if r == rows - 1:
            rans = self.recurr(dungeon, r, c + 1, dp) - dungeon[r][c]
            if rans <= 0:
                dp[r][c] = 1
                return 1
            dp[r][c] = rans
            return rans

        if c == cols - 1:
            cans = self.recurr(dungeon, r + 1, c, dp) - dungeon[r][c]
            if cans <= 0:
                dp[r][c] = 1
                return 1
            dp[r][c] = cans
            return cans

        ans = min(self.recurr(dungeon, r + 1, c, dp), self.recurr(dungeon, r, c + 1, dp)) - dungeon[r][c]
        if ans <= 0:
            dp[r][c] = 1
            return 1
        dp[r][c] = ans
        return ans

    def calculateMinimumHP(self, dungeon):
        rows, cols = len(dungeon), len(dungeon[0])
        dp = [[float('inf')] * cols for _ in range(rows)]
        return self.recurr(dungeon, 0, 0, dp)
```

[ ]:

```
#### https://leetcode.com/problems/minimum-path-cost-in-a-grid/description/

class Solution:
    def __init__(self):
        self.dp = [[-1 for _ in range(51)] for _ in range(51)]

    def recurr(self, grid, cost, idx, pos):
        m, n = len(grid), len(grid[0])

        if idx == m - 1:
            return grid[idx][pos]
        if idx >= m:
            return 0

        if self.dp[idx][pos] != -1:
            return self.dp[idx][pos]

        ans = float('inf')
        for i in range(n):
            val = grid[idx][pos]
            mc = val + cost[val][i]
            ans = min(ans, mc + self.recurr(grid, cost, idx + 1, i))

        self.dp[idx][pos] = ans
        return ans

    def minPathCost(self, grid, moveCost):
        self.dp = [[-1 for _ in range(51)] for _ in range(51)]
        ans = float('inf')
        n = len(grid[0])
        for i in range(n):
            ans = min(ans, self.recurr(grid, moveCost, 0, i))

        return ans
```

[ ]:

```
#### https://leetcode.com/problems/cherry-pickup/description/
```

[ ]:

```
#### https://leetcode.com/problems/
 ↪number-of-ways-to-reach-a-position-after-exactly-k-steps/description/
```

[ ]:

```
#### https://leetcode.com/problems/paths-in-matrix-whose-sum-is-divisible-by-k/
 ↪description/
```

[ ]:

```
#### https://leetcode.com/problems/last-stone-weight-ii/description/
```

[ ]:

```
#### https://leetcode.com/problems/best-time-to-buy-and-sell-stock/description/

def maxProfit(self, prices: List[int]) -> int:
        if not prices:
                return 0

        maxProfit = 0
        minPurchase = prices[0]
        for i in range(1, len(prices)):
                maxProfit = max(maxProfit, prices[i] - minPurchase)
                minPurchase = min(minPurchase, prices[i])
        return maxProfit
```

```
#### https://leetcode.com/problems/best-time-to-buy-and-sell-stock-ii/
 ↪description/
#### You may complete as many transactions as you like (i.e., buy one and sell␣
 ↪one share of the stock multiple times).

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        ans, sz = 0, len(prices)

        for i in range(1, sz, 1):
            if prices[i-1] < prices[i]:
                ans += prices[i] - prices[i-1]

        return ans

#### https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iii/
 ↪description/
#### https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iv/
 ↪description/

"""
class Solution {
public:
    // int dp[100001][2][]
    int maxProfit(vector<int>& prices) {
        int sz = prices.size(), k = 2;
        vector<vector<vector<int>>> dp(sz, vector<vector<int>>(k,␣
 ↪vector<int>(2, -1)));
        return recurr(prices, 0, 0, 1, k, dp);
    }

    int recurr(vector<int> &prices, int idx, int cnt, int type, int k,␣
 ↪vector<vector<vector<int>>> &dp) {
        if (idx >= prices.size() or cnt >= k) {
            return 0;
        }

        if (dp[idx][cnt][type] != -1) {
            return dp[idx][cnt][type];
        }

        if (type) {
            dp[idx][cnt][type] = max(-prices[idx]+recurr(prices, idx+1, cnt, !
 ↪type, k, dp), recurr(prices, idx+1, cnt, type, k, dp));
        } else {
            dp[idx][cnt][type] = max( prices[idx]+recurr(prices, idx+1, cnt+1, !type, k, dp), recurr(prices, idx+1, cnt, type, k, dp));
        }

        return dp[idx][cnt][type];
    }
};
"""

#### https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-cooldown/description/

"""
class Solution {
public:
    vector<vector<int>> dp;
    int maxProfit(vector<int>& prices) {
        int n = prices.size();
        dp.resize(2, vector<int>(n, -1));
        return recurr(prices, 0, true);
    }

    int recurr(vector<int> & prices, int day, bool canBuy) {
        if (day >= prices.size()) {
            return 0;
        }

        if (dp[canBuy ? 0 : 1][day] != -1) {
            return dp[canBuy ? 0 : 1][day];
        }

        int ans = recurr(prices, day+1, canBuy);

        if (canBuy) {
            ans = max(ans, -prices[day] + recurr(prices, day+1, !canBuy));
        } else {
            ans = max(ans, prices[day] + recurr(prices, day+2, !canBuy));
        }

        dp[canBuy ? 0 : 1][day] = ans;

        return ans;
    }
};
"""
```

```
#### https://leetcode.com/problems/
 ↪best-time-to-buy-and-sell-stock-with-transaction-fee/description/

"""
class Solution {
public:
    int maxProfit(vector<int>& prices, int fee) {
        int sz = prices.size();
        vector<vector<int>> dp(sz, vector<int>(2, -1));
        return recurr(prices, 0, 1, fee, dp);
    }

    int recurr(vector<int> &prices, int idx, int type, int fee,␣
 ↪vector<vector<int>> &dp) {
        if (idx >= prices.size()) {
            return 0;
        }

        if (dp[idx][type] != -1) {
            return dp[idx][type];
        }

        if (type) {
            dp[idx][type] = max(-prices[idx]-fee+recurr(prices, idx+1, !type,␣
 ↪fee, dp), recurr(prices, idx+1, type, fee, dp));
        } else {
            dp[idx][type] = max(prices[idx]+recurr(prices, idx+1, !type, fee,␣
 ↪dp), recurr(prices, idx+1, type, fee, dp));
        }

        return dp[idx][type];
    }
};
"""
```

[ ]:

```
#### https://leetcode.com/problems/coin-change/description/

"""
Input: coins = [1,2,5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1
"""

class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        @lru_cache(None)
        def dp(i, amount):
            if amount == 0:
                return 0
            if i == -1:
                return math.inf

            ans = dp(i-1, amount)  # Skip ith coin
            if amount >= coins[i]:  # Used ith coin
                ans = min(ans, dp(i, amount - coins[i]) + 1)
            return ans

        n = len(coins)
        ans = dp(n-1, amount)
        return ans if ans != math.inf else -1


#### https://leetcode.com/problems/coin-change-ii/description/

class Solution:
    def change(self, amount: int, coins: list[int]) -> int:
        dp = [[None for _ in range(amount + 1)] for _ in range(len(coins))]
        return self.dfs(coins, 0, amount, dp)

    def dfs(self, coins: list[int], i: int, amount: int, dp: list[list[None]]) -> int:
        if amount == 0:
            return 1
        if i == len(coins):
            return 0
        if dp[i][amount] is not None:
            return dp[i][amount]
        ans = self.dfs(coins, i + 1, amount, dp)  # skip ith coin
        if amount >= coins[i]:
            ans += self.dfs(coins, i, amount - coins[i], dp)  # use ith coin
        dp[i][amount] = ans
        return ans
```

[ ]:

```
#### https://leetcode.com/problems/maximum-value-of-k-coins-from-piles/description/
```

[ ]:

```
#### https://leetcode.com/problems/target-sum/description/
```