# Notebook

August 6, 2024

**Identify the problem if it falls other which category**

- 0/1 Knapsack
- Unbounded Knapsack
- Shortest Path (eg: Unique Paths I/II)
- Fibonacci Sequence (eg: House Thief, Jump Game)
- Longest Common Substring/Subsequeunce

[ ]:
```
#### https://github.com/doocs/leetcode/blob/main/solution/1000-1099/1062.
 ↪Longest%20Repeating%20Substring/README_EN.md
#### Longest repeating substring - non overlappping
#### Google

"""
Input: s = "abbaba"
Output: 2
Explanation: The longest repeating substrings are "ab" and "ba", each of which␣
 ↪occurs twice.
"""

class Solution:
    def longestRepeatingSubString(s):
        n = len(s)
        dp = [[0]*n for _ in range(n)]
        ans = 0
        for i in range(n):
            for j in range(i+1):
                if s[i] == s[j]:
                    dp[i][j] = dp[i-1][j-1] + 1 if i > 0 and j > 0 else 1
                    ans = max(ans, dp[i][j])

        return ans

# Time complexity: O(n^2)
# Space complexity: O(n^2)
```

[ ]:
```
#### https://www.geeksforgeeks.org/problems/mobile-numeric-keypad5456/1
#### Flipkart
```

[ ]:
```
#### https://leetcode.com/problems/maximum-total-damage-with-spell-casting/
 ↪description/
```

[ ]:
```
#### Find the length of longest increasing subsequence such that the difference␣
 ↪between consecutive elements in LIS is an increasing sequence

"""
private static Map<String, Integer> map;

public static void main(String[] args) {
    map = new HashMap<>();
    int[] nums = new int[]{1, 2, 3, 4, 5, 6};
    System.out.println(lisWithLisDelta(0, -1, 0, nums));
    nums = new int[]{1, 11, 12, 14};
    map = new HashMap<>();
    System.out.println(lisWithLisDelta(0, -1, 0, nums));
}

private static int lisWithLisDelta(int index, int prevIndex, int prevDelta,␣
 ↪int[] nums){
    if(index == nums.length)
        return 0;
    String key = index +"_"+ prevIndex + "_" + prevDelta;

    if(map.containsKey(key)){
        return map.get(key);
    }


    int result = 0;

    if(prevIndex != -1 && (nums[prevIndex] >= nums[index] || prevDelta >=␣
 ↪nums[index] - nums[prevIndex])){
        result = lisWithLisDelta(index + 1, prevIndex, prevDelta, nums);
    } else{
        result = Math.max(1 + lisWithLisDelta(index + 1, index, prevIndex == -1␣
 ↪? 0 : nums[index] - nums[prevIndex], nums), lisWithLisDelta(index + 1,␣
 ↪prevIndex, prevDelta, nums));
    }
        map.put(key, result);

    return result;
}
"""
```

[ ]:
```
# https://leetcode.com/problems/minimum-falling-path-sum/description/
# Input: matrix = [[2,1,3],[6,5,4],[7,8,9]]  Output: 13
```

```python
# go from top to bottom, find the minimum sum path where you can move only left, right or down

class Solution:
    def minFallingPathSum(self, A: List[List[int]]) -> int:
        for i in range(1, len(A)):
            for j in range(len(A)):
                A[i][j] += min(A[i-1][j], A[i-1][j-1] if j > 0 else float('inf'), A[i-1][j+1] if j < len(A) - 1 else float('inf'))
        return min(A[-1])

# https://leetcode.com/problems/minimum-falling-path-sum-ii/description/
# A falling path with non-zero shifts is a choice of exactly one element from each row of grid such that no two elements chosen in adjacent rows are in the same column.

def min_falling_path_sum(grid):
    m, n = len(grid), len(grid[0])
    dp = [[0] * n for _ in range(m)]

    for i in range(n):
        dp[0][i] = grid[0][i]

    for i in range(1, m):
        for j in range(n):
            ans = sys.maxsize
            for k in range(n):
                if j == k:
                    continue
                ans = min(ans, dp[i-1][k] + grid[i][j])
            dp[i][j] = ans

    return min(dp[m-1])
```

[ ]:
```python
# https://leetcode.com/problems/target-sum/description/
# Input: nums = [1,1,1,1,1], target = 3 Output: 5
# Explanation: There are 5 ways to assign symbols to make the sum of nums be target 3.
# -1 + 1 + 1 + 1 + 1 = 3    +1 - 1 + 1 + 1 + 1 = 3    +1 + 1 - 1 + 1 + 1 = 3
# +1 + 1 + 1 - 1 + 1 = 3    +1 + 1 + 1 + 1 - 1 = 3

# Fall under 0/1 knapsack problem
#
class Solution:
    def findTargetSumWays(self, nums: List[int], target: int) -> int:
        memo = {}
        def recurr(nums, target, index, curr_sum):
```

```
            if (index, curr_sum) in memo: return memo[(index, curr_sum)]
            if index < 0 and curr_sum == target: return 1
            if index < 0 : return 0

            positive = recurr(nums, target, index-1, curr_sum + nums[index])
            negative = recurr(nums, target, index-1, curr_sum - nums[index])

            memo[(index, curr_sum)] = positive + negative
            return memo[(index, curr_sum)]
        return recurr(nums, target, len(nums)-1, 0)
```

[ ]:
```
# https://leetcode.com/problems/ways-to-express-an-integer-as-sum-of-powers/
 ↪description/
# Input: n = 4, x = 1   Output: 2
# - n = 4^1 = 4.    - n = 3^1 + 1^1 = 4.

# Time Complexity - O(N^2)
class Solution:
    def numberOfWays(self, n: int, x: int) -> int:
        memo = {}
        mod = 10**9 + 7
        def recurr(n, x, num):
            if (n < 0) : return 0
            if (n == 0) : return 1
            if (num**x > n) :  return 0
            if (n, num) in memo: return memo[(n, num)]
            temp = num**x

            pick = recurr(n-temp, x, num+1)
            skip = recurr(n, x, num+1)
            memo[(n, num)] = (skip % mod + pick % mod) % mod
            return memo[(n, num)]
        return recurr(n, x, 1)
```

[ ]:
```
# https://leetcode.com/problems/filling-bookcase-shelves/description/

# Time Complexity - O(N * M) with memoisation where N is the number of books␣
 ↪and M is the shelf width
# Time Complexity - O(2 ^ N) without memoisation where N is the number of books
# Space Complexity - O(N * M)

class Solution:
    def minHeightShelves(self, books: List[List[int]], shelfWidth: int) -> int:
        n, memo = len(books), {}
        def recurr(idx, width, height):
            if idx >= n: return height
            if (idx, width, height) in memo: return memo[(idx, width, height)]
```

```
            curr_width, curr_height = books[idx]

            # we start a new shelf and add height of previous shelf
            take = height + recurr(idx+1, curr_width, curr_height)
            skip = float('inf')

            # we add the current book to the previous shelf and take max height
            if (width + curr_width <= shelfWidth):
                skip = recurr(idx+1, curr_width + width, max(height,␣
↪curr_height))
            memo[(idx, width, height)] = min(take, skip)
            return memo[(idx, width, height)]
        return recurr(0, 0, 0)
```