Dynamic Programming

Mustafa Muhammad

25 September 2021

List of Questions:

0-1 Knapsack

1. Subset Sum
2. Equal Sum Partition
3. Count of Subset Sum with given Sum
4. Minimum Subset Sum Difference
5. Count number of subsets with given difference
6. Target Sum

Unbounded Knapsack

1. Rod Cutting Problem
2. Max coin change problem
3. Min number of coins

Longest Common Subsequence (LCS)

1. Longest common substring
2. Shortest common supersequence
3. Min number of insertion deletion to convert string a to b
4. Longest palindromic subsequence
5. Min number of deletions to make string into palindrome
6. Print shorest common supersequence
7. Longest repeating subsequence
8. Sequence pattern matching
9. Min number of insertions in a string to make it a palindrome

Dp on Trees

1. Diameter of binary tree
2. Max path sum from any node
3. Max path sum from leaf to leaf

1 0-1 Knapsack

You are given weights and values of N items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

```
class Solution:
    def __init__(self):
        self.memo={}

    def knapSack(self,W, wt, val, n):
        key = (W, n)
        if n == 0 or W == 0:
            return 0

        if key in self.memo:
            return self.memo[key]

        #Capacity less than the item's weight, so skip it
        if wt[n-1] > W:
            self.memo[key] = self.knapSack(W, wt, val, n-1)
            return self.memo[key]

        #Optimization step, to choose the maximum
        max_val = max(val[n-1] + self.knapSack(W-wt[n-1], wt, val, n-1),
        self.knapSack(W, wt, val, n-1))

        self.memo[key] = max_val

        return self.memo[key]
```

Input:

N = 3

W = 4

values[] = 1,2,3

weight[] = 4,5,1

Output: 3

2 Subset Sum

Given an array of non-negative integers, and a value sum, determine if there is a subset of the given set with sum equal to given sum.

```
class Solution:
    def __init__(self):
        self.memo = {}

    def isSubsetSum (self, N, arr, sum):
        key = (N, sum)

        if key in self.memo:
            return self.memo[key]

        if N == 0 and sum != 0:
            return False

        if sum == 0:
            return True


        if arr[N-1] > sum:
            self.memo[key] = self.isSubsetSum(N-1, arr, sum)
            return self.memo[key]

        self.memo[key] = self.isSubsetSum(N-1, arr, sum-arr[N-1])
        or self.isSubsetSum(N-1, arr, sum)

        return self.memo[key]
```

Input:

N = 6

arr[] = 3, 34, 4, 12, 5, 2

sum = 9

Output: 1

Explanation: Here there exists a subset with sum = 9, 4+3+2 = 9.

3 Subset sum – Using backtracking and memoization

```
class Solution:
    def isSubsetSum (self, N, arr, sum):
        memo = {}
        def combinations(N, arr, target, curr_sum):
                        key = (N, curr_sum)
                        if key in memo:
                                            return memo[key]
            if target == curr_sum:
                return True
            if N == 0:
                return False

            curr_sum += arr[N-1]
            a = combinations(N-1, arr, target, curr_sum)
            curr_sum -= arr[N-1]
            b = combinations(N-1, arr, target, curr_sum)

            memo[key] = a or b
            return memo[key]

        return combinations(N, arr, sum, 0)
```

In practice the complexity should be the same as dynamic programming.

4 Equal Sum Partition

Crux of the solution is the fact that we cannot partition an array into equal parts which has an odd sum. If the array has an even sum, all we need to do is to call the boolean subsetSum on half of the total sum of the array.

```
class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        total = sum(nums)

        if total % 2 == 1:
            return False

        memo = {}

        def subset_sum(i, nums, target):
            key = (i, target)
            if key in memo:
                return memo[key]
            if target == 0:
                return True

            if i == 0 and target != 0:
                return False

            if nums[i-1] > target:
                memo[key] = subset_sum(i-1, nums, target)
                return memo[key]

            memo[key] = subset_sum(i-1, nums, target-nums[i-1])
            or subset_sum(i-1, nums, target)

            return memo[key]

        return subset_sum(len(nums), nums, int(total/2))
```

Given a non-empty array nums containing only positive integers, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

Input: nums = [1,5,11,5]

Output: true

Explanation: The array can be partitioned as [1, 5, 5] and [11].

5 Count Number of subsets that add up to a target

This problem can easily be solved by using backtracking and generating all the possible subsets. However that will lead to exponential time complexity.

Dynamic programming is an optimization technique used to reduce time complexity by building up from smaller / previous outputs.

```
class Solution:
    def perfectSum(self, arr, n, target):
        memo = {}
        def helper(arr, pos, target):
            key = (pos, target)
            if target == 0:
                return 1
            if pos == 0:
                return 0
            if key in memo:
                return memo[key]
            else:
                if arr[pos-1] > target:
                    memo[key] = helper(arr, pos-1, target)
                    return memo[key]
                else:
                    memo[key] = helper(arr, pos-1, target-arr[pos-1])
                    + helper(arr, pos-1, target)
                    return memo[key]
            return memo[key]
```

6 Minimum Subset Sum Difference

Given a set of integers, the task is to divide it into two sets S1 and S2 such that the absolute difference between their sums is minimum.

If there is a set S with n elements, then if we assume Subset1 has m elements, Subset2 must have n-m elements and the value of abs(sum(Subset1) – sum(Subset2)) should be minimum.

```
class Solution:
    def minDifference(self, arr, n):
        memo = {}

        def subset_sum(arr, i, target):
            key = (i, target)
            if target == 0:
                return True
            if i == 0:
                return False
            if key in memo:
                return memo[key]
            if arr[i-1] > target:
                memo[key] = subset_sum(arr, i-1, target)
                return memo[key]
            memo[key] = subset_sum(arr, i-1, target-arr[i-1])
            or subset_sum(arr, i-1, target)
            return memo[key]

        Range = sum(arr)
        ans = float('inf')
        for i in range(0, int(Range/2)+1):
            if subset_sum(arr, len(arr) , i):
                ans = min(ans, Range - 2*i)
        return ans
```

This is a hard question to grasp on the first try. The required answer is two subsets with the minimum difference. They can be labelled as s1 and s2. We also know that the answer lies between the range 0, sum(given array).

Another thing that we know is that we can split the range between s1 and s2. With s1 on the left and s2 on the right.

Since Range - s1 gives us s2. We can minimize the problem to Range -2 s1. From there on we create a loop from 0 to Range/2 and find the minimum value that satisfies the requirement of min(ans, Range - 2*s1).

7 Count number of subsets with given difference

```
class Solution:
    def count_number_of_subsets_with_difference(self, arr, diff):

        sum_of_arr = sum(arr)

        target = (diff+sum_of_arr)/2

        memo = {}

        def subset_sum(arr, i, target):
            key = (i, target)

            if target == 0:
                return 1

            if i == 0:
                return 0

            if key in memo:
                return memo[key]

            if arr[i-1] > target:
                memo[key] = subset_sum(arr, i-1, target)
                return memo[key]

            memo[key] = subset_sum(arr, i-1, target-arr[i-1])
            + subset_sum(arr, i-1, target)
            return memo[key]

        return subset_sum(arr, len(arr), target)
```

Since we know the difference, we can model two equations. S2-S1 = diff, S1 + S2 = sum(arr)

Hence target = (diff + sum(arr))/2

We can run subset sum on the target and get the count.

8 Target Sum

Same problem as Count number of subsets with difference, just with different wording.

```
class Solution:
    def findTargetSumWays(self, nums: List[int], target: int) -> int:
        if sum(nums) < target or (sum(nums)-target)%2:
            return 0
        s1 = (sum(nums)+target)/2
        memo = {}
        def subset_count(arr, i, target):
            key = (i, target)
            if i == 0:
                return 0 if target else 1
            if key in memo:
                return memo[key]
            if arr[i-1] > target:
                memo[key] = subset_count(arr, i-1, target)
                return memo[key]

            memo[key] = subset_count(arr, i-1, target-arr[i-1])
            + subset_count(arr, i-1, target)
            return memo[key]
        return subset_count(nums, len(nums), s1)
```

You are given an integer array nums and an integer target.

You want to build an expression out of nums by adding one of the symbols '+' and '-' before each integer in nums and then concatenate all the integers.

For example, if nums = [2, 1], you can add a '+' before 2 and a '-' before 1 and concatenate them to build the expression "+2-1". Return the number of different expressions that you can build, which evaluates to target.

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

9 Unbounded KnapSack

In an unbounded knapsack we are allowed to take multiple occurences of an item. If an item is selected once, it can be taken again. If it is ignored the first time, it will not be taken in subsequent iterations as well.

There is only a minor change in the coding style for unbounded knapsack.

```
# Minor change at the end !

return knapsack(arr, i, target-arr[i-1]) or knapsack(arr, i-1, target)
```

10 Rod cutting problem

```
class Solution:

    def __init__(self):
        self.memo = {}

    def unbounded_knapsack(self, items, weight, capacity, index):
        key = (index, capacity)
        if key in self.memo:
            return self.memo[key]

        if index == 0 or capacity == 0:
            self.memo[key] = 0
            return self.memo[key]

        if weight[index-1] > capacity:
            self.memo[key] = self.unbounded_knapsack(items, weight, capacity, index-1)
            return self.memo[key]

        self.memo[key] = max(items[index-1]+self.unbounded_knapsack(items,weight,capaci
            self.unbounded_knapsack(items,weight,capacity,index-1))
        return self.memo[key]
```

11 Coin change problem (max number of ways)

Given a value N, find the number of ways to make change for N cents, if we have infinite supply of each of S = S1, S2, .. , SM valued coins.

```
class Solution:
    def count(self, S, m, n):
        i = m
        target = n
        arr = S
        memo = {}
        def subset_sum(arr, i, target):
            key = (i, target)
            if target == 0:
                return 1

            if i == 0:
                return 0

            if key in memo:
                return memo[key]

            if arr[i-1] > target:
                memo[key] = subset_sum(arr, i-1, target)
                return memo[key]

            memo[key] = subset_sum(arr, i, target-arr[i-1]) + subset_sum(arr, i-1, targ
            return memo[key]
        return subset_sum(arr, i, target)
```

Input:

n = 4 , m = 3

S[] = 1,2,3

Output: 4

Explanation: Four Possible ways are:

1,1,1,1,1,1,2,2,2,1,3.

12 Minimum number of coins

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.

```
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        memo = {}
        def unbounded_knapsack(arr, i, target):
            key = (i, target)
            if target == 0:
                return 0

            if i <= 0 and target > 0:
                return float('inf')

            if key in memo:
                return memo[key]

            if arr[i-1] > target:
                memo[key] = unbounded_knapsack(arr, i-1, target)
                return memo[key]

            memo[key] = min(1+unbounded_knapsack(arr,i,target-arr[i-1]),
                            unbounded_knapsack(arr,i-1,target))
            return memo[key]
        res = unbounded_knapsack(coins, len(coins), amount)
        if res == float('inf'):
            return -1
        return res
```

Input: coins = [1,2,5], amount = 11

Output: 3

Explanation: 11 = 5 + 5 + 1

13 Longest Common Subsequence

```
def longestCommonSubsequence(self, text1: str, text2: str)
 -> int:
        memo = {}
        def lcs(s1, s2, i, j):
            key = (i, j)

            if i <=0 or j<=0:
                return 0

            if key in memo:
                return memo[key]

            if s1[i-1] == s2[j-1]:
                memo[key] = 1+lcs(s1, s2, i-1, j-1)
                return memo[key]

            memo[key] = max(lcs(s1, s2, i-1, j), lcs(s1, s2, i, j-1))
            return memo[key]

        return lcs(text1, text2, len(text1), len(text2))
```

Given two strings text1 and text2, return the length of their longest common subsequence. If there is no common subsequence, return 0.

A subsequence of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

For example, "ace" is a subsequence of "abcde". A common subsequence of two strings is a subsequence that is common to both strings.

Example 1:

Input: text1 = "abcde", text2 = "ace"

Output: 3

Explanation: The longest common subsequence is "ace" and its length is 3.

14 Longest Common Substring

```
class Solution:
    def __init__(self):
        self.res = 0
    def longestCommonSubstr(self, S1, S2, n, m):
        def helper(s1, s2, n, m):

            if n == 0 or m == 0:
                return 0

            if s1[n-1] == s2[m-1]:
                a  = 1+helper(s1, s2, n-1, m-1)
                self.res = max(self.res, a)
                return a

            helper(s1, s2, n-1, m)
            helper(s1, s2, n, m-1)

            return 0

        helper(S1, S2, n, m)

        return self.res
```

Different from LCS in the sense that we return only when there is a match

15 Printing Longest Common SubSequence

```
class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        memo = {}
        res = []
        def lcs(s1, s2, i, j):
            key = (i, j)

            if i <=0 or j<=0:
                return 0

            if key in memo:
                return memo[key]

            if s1[i-1] == s2[j-1]:
                res.append(s1[i-1])
                memo[key] = 1+lcs(s1, s2, i-1, j-1)
                return memo[key]

            memo[key] = max(lcs(s1, s2, i-1, j), lcs(s1, s2, i, j-1))
            return memo[key]
        lcs(text1, text2, len(text1), len(text2))
        print(res)
        return
```

16 Shortest Common SuperSequence (Finding Length)

```
class Solution:
    def shortestCommonSupersequence(self, X, Y, m, n):
        memo = {}
        def helper(s1, s2, i, j):
            key = (i, j)
            if i<=0 or j<=0:
                return 0
            if key in memo:
                return memo[key]
            if s1[i-1] == s2[j-1]:
                memo[key] = 1+helper(s1, s2, i-1, j-1)
                return memo[key]
            memo[key] = max(helper(s1, s2, i-1, j), helper(s1, s2, i, j-1))
            return memo[key]
        return m+n - helper(X, Y, m, n)
```

17 Printing Shortest Common SuperSequence

```
class Solution:
    def shortestCommonSupersequence(self, str1: str, str2: str) -> str:
        arr = []
        memo = {}
        def helper(s1, s2, i, j):
            key = (i, j)
            if not i and not j:
                return ""
            if i == 0:
                return s2[:j]
            if j == 0:
                return s1[:i]
            if key in memo:
                return memo[key]
            if s1[i-1] == s2[j-1]:
                memo[key] = helper(s1, s2, i-1, j-1) + s1[i-1]
            else:
                a = helper(s1, s2, i-1, j) + s1[i-1]
                b = helper(s1, s2, i, j-1) + s2[j-1]
                if len(a) <= len(b):
                    memo[key] = a
                else:
                    memo[key] = b
            return memo[key]
        return helper(str1, str2, len(str1), len(str2))
```

Given two strings str1 and str2, return the shortest string that has both str1 and str2 as subsequences. If there are multiple valid strings, return any of them.

A string s is a subsequence of string t if deleting some number of characters from t (possibly 0) results in the string s.

Example 1:

Input: str1 = "abac", str2 = "cab"

Output: "cabac"

Explanation: str1 = "abac" is a subsequence of "cabac" because we can delete the first "c".

str2 = "cab" is a subsequence of "cabac" because we can delete the last "ac".

The answer provided is the shortest such string that satisfies these properties.

18 Minimum Number of insertions/deletions to convert string a to b

```
class Solution:
    def minOperations(self, s1, s2):
        # code here
        memo = {}
        def lcs(s1, s2, i, j):
            key = (i, j)
            if i == 0 or j == 0:
                return 0
            if key in memo:
                return memo[key]
            if s1[i-1] == s2[j-1]:
                memo[key] = 1 + lcs(s1, s2, i-1, j-1)
                return memo[key]
            memo[key] = max(lcs(s1, s2, i-1, j), lcs(s1, s2, i, j-1))
            return memo[key]
        return len(s1) +len(s2) - 2*lcs(s1, s2, len(s1), len(s2))
```

Number of deletion = len(s1) - lcs Number of insetion = len(s2) - lcs

Hence total = len(s1) + len(s2) - 2*lcs

19 Longest Palindromic Subsequence

```
class Solution:
    def longestPalindromeSubseq(self, s: str) -> int:
        memo = {}
        def lcs(s1, s2, i, j):
            key = (i, j)

            if i == 0 or j == 0:
                return 0

            if key in memo:
                return memo[key]

            if s1[i-1] == s2[j-1]:
                memo[key] = 1+ lcs(s1, s2, i-1, j-1)
                return memo[key]

            memo[key] = max(lcs(s1, s2, i-1, j), lcs(s1, s2, i, j-1))
            return memo[key]

        return lcs(s, s[::-1], len(s), len(s))
```

LCS with s and reverse of s.

20 Min number of deletion in string to make a palindrome

```
class Solution:
    def minimumNumberOfDeletions(self,S):
        # code here
        memo = {}
        def lcs(s1, s2, i, j):
            key = (i, j)
            if i == 0 or j == 0:
                return 0

            if key in memo:
                return memo[key]

            if s1[i-1] == s2[j-1]:
                memo[key] = 1+ lcs(s1, s2, i-1, j-1)
                return memo[key]

            memo[key] = max(lcs(s1, s2, i-1, j), lcs(s1, s2, i, j-1))
            return memo[key]

        return len(S) - lcs(S, S[::-1], len(S), len(S))
```

Min deletions: len(s) - len of palindromic subsequence

21 Longest Repeating Subsequence

```
class Solution:
    def LongestRepeatingSubsequence(self, str):
        # Code here
        memo = {}
        def lcs(s1, s2, i, j):
            key = (i, j)
            if i == 0 or j == 0:
                return 0

            if key in memo:
                return memo[key]

            # i not eq j
            if s1[i-1] == s2[j-1] and i != j:
                memo[key] = 1+ lcs(s1, s2, i-1, j-1)
                return memo[key]

            memo[key] = max(lcs(s1, s2, i-1, j), lcs(s1, s2, i, j-1))
            return memo[key]

        return lcs(str, str, len(str), len(str))
```

Given a string str, find the length of the longest repeating subsequence such that it can be found twice in the given string. The two identified subsequences A and B can use the same ith character from string str if and only if that ith character has different indices in A and B.

Example 2:

Input:

str = "aab"

Output: 1

Explanation:

The longest reapting subsequenece is "a".

22 Sequence Pattern Matching

Q: Does A exist in B in the same order

A: "AXY"

B: "AXYZ"

Return True since the LCS of A and B is equal to A.

23 Min Number of Insertion to make a String a Palindrome

```
class Solution:
    def countMin(self,S):
        # code here
        memo = {}
        def isPalindrome(s):
            return s == s[::-1]

        if isPalindrome(S):
            return 0

        def lcs(s1, s2, i, j):
            key = (i, j)
            if i == 0 or j == 0:
                return 0

            if key in memo:
                return memo[key]

            if s1[i-1] == s2[j-1]:
                memo[key] = 1+ lcs(s1, s2, i-1, j-1)
                return memo[key]

            memo[key] = max(lcs(s1, s2, i-1, j), lcs(s1, s2, i, j-1))
            return memo[key]

        return len(S) - lcs(S, S[::-1], len(S), len(S))
```

Min number of insetions is equal to the min number of deletions. Hence its the same problem asked in a different way from before.

24 DP On Trees – Diameter of Tree

```
class Solution:
    def diameter(self,root):
        res = [0]
        def helper(root):
            if root == None:
                return 0

            l = helper(root.left)
            r = helper(root.right)

            temp = max(l, r) + 1

            res[0] = max(l+r+1, res[0])

            return temp
        helper(root)
        return res[0]
```

We have two choices at every root, either we pass the left or right branch branch including the root or the the answer passes through the current root itself in an upside down parabola shape.

25 Max Path Sum from any node to any node

```
class Solution:
    def maxPathSum(self, root: Optional[TreeNode]) -> int:
        res = [float('-inf')]
        def helper(root):
            if root == None:
                return 0

            l = helper(root.left)
            r = helper(root.right)

            temp = max(max(l,r)+root.val, root.val)
            ans = max(temp ,l+r+root.val )
            res[0] = max(ans, res[0])

            return temp

        helper(root)
        return res[0]
```

We have two choices as before, choose the max of both l,r and add value of root or just the root.val to avoid negatives.

26 Max Path Sum from leaf node to leaf node

```
class Solution:
    def maxPathSum(self, root):
        res = [float('-inf')]

        def helper(root):
            if root == None:
                return 0

            l = helper(root.left)
            r = helper(root.right)

            temp = max(max(l, r)+root.data, root.data)
            if root.left == None and root.right == None:
                temp = max(temp, root.data)
            ans = max(l+r+root.data, temp)
            res[0] = max(res[0], ans)
            return temp

        helper(root)
        return res[0]
```