DYNAMIC PROGRAMMING

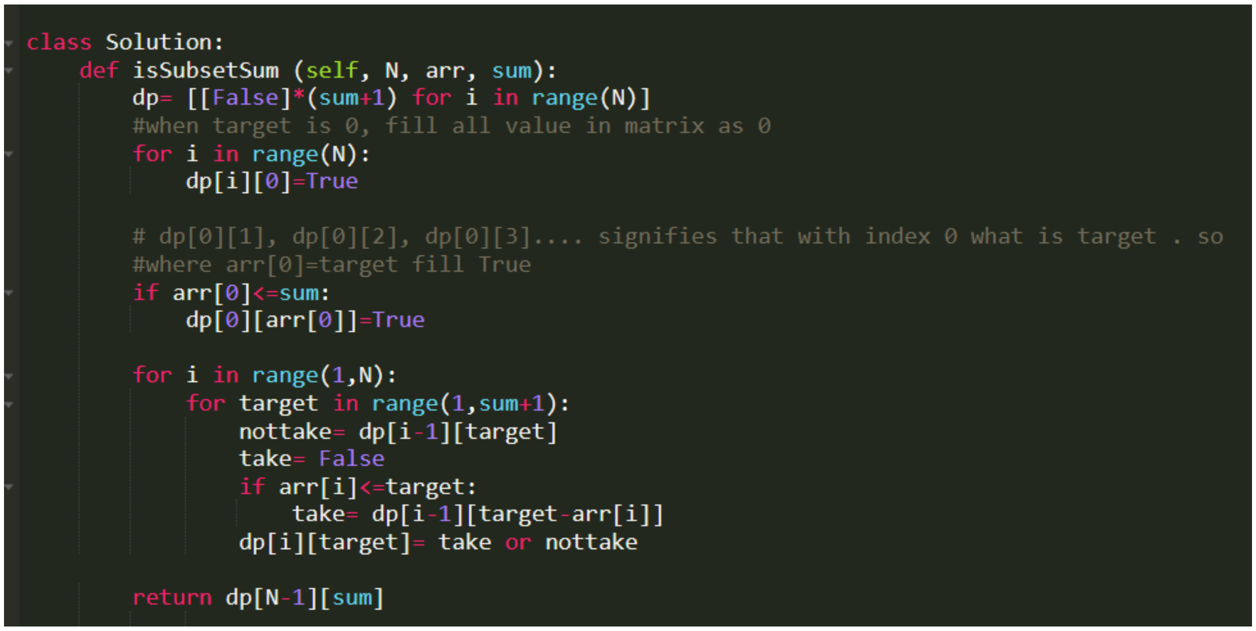
Problem on Subsequences

—-------------------------------------

Q: Subset sum equals to target:
Using memoization:

```python
class Solution:
    def isSubsetSum (self, N, arr, sum):
        def f(ind,target):

            if target==0:
                return True
            if ind==0:
                if arr[0]==target:
                    return True
                else:
                    return False
            if dp[ind][target]!=-1:
                return dp[ind][target]
            nottake= f(ind-1,target)
            take= False
            if arr[ind]<=target:
                take= f(ind-1,target-arr[ind])
            dp[ind][target]= take or nottake
            return dp[ind][target]

        dp= [[-1]*(sum+1) for i in range(N)]
        return f(N-1,sum)
```

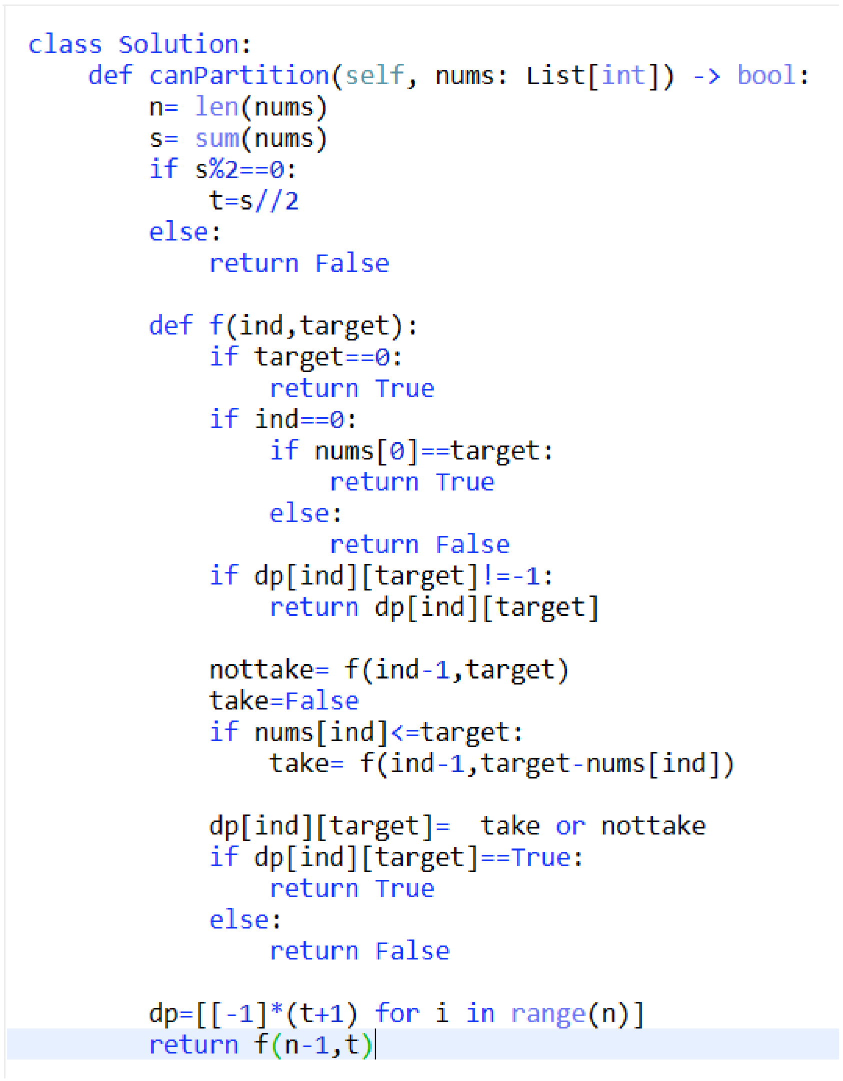
Using tabulation:



```
class Solution:
    def isSubsetSum (self, N, arr, sum):
        dp= [[False]*(sum+1) for i in range(N)]
        #when target is 0, fill all value in matrix as 0
        for i in range(N):
            dp[i][0]=True

        # dp[0][1], dp[0][2], dp[0][3].... signifies that with index 0 what is target . so
        #where arr[0]=target fill True
        if arr[0]<=sum:
            dp[0][arr[0]]=True

        for i in range(1,N):
            for target in range(1,sum+1):
                nottake= dp[i-1][target]
                take= False
                if arr[i]<=target:
                    take= dp[i-1][target-arr[i]]
                dp[i][target]= take or nottake

        return dp[N-1][sum]
```

Q: Partition equal subset sum:

This question is same as subset sum = target as the set can be partition only if sum of all element divide by 2 . if not then it definately not able to partition. If divide by 2 then need to check weather it can be possible to partition or not.



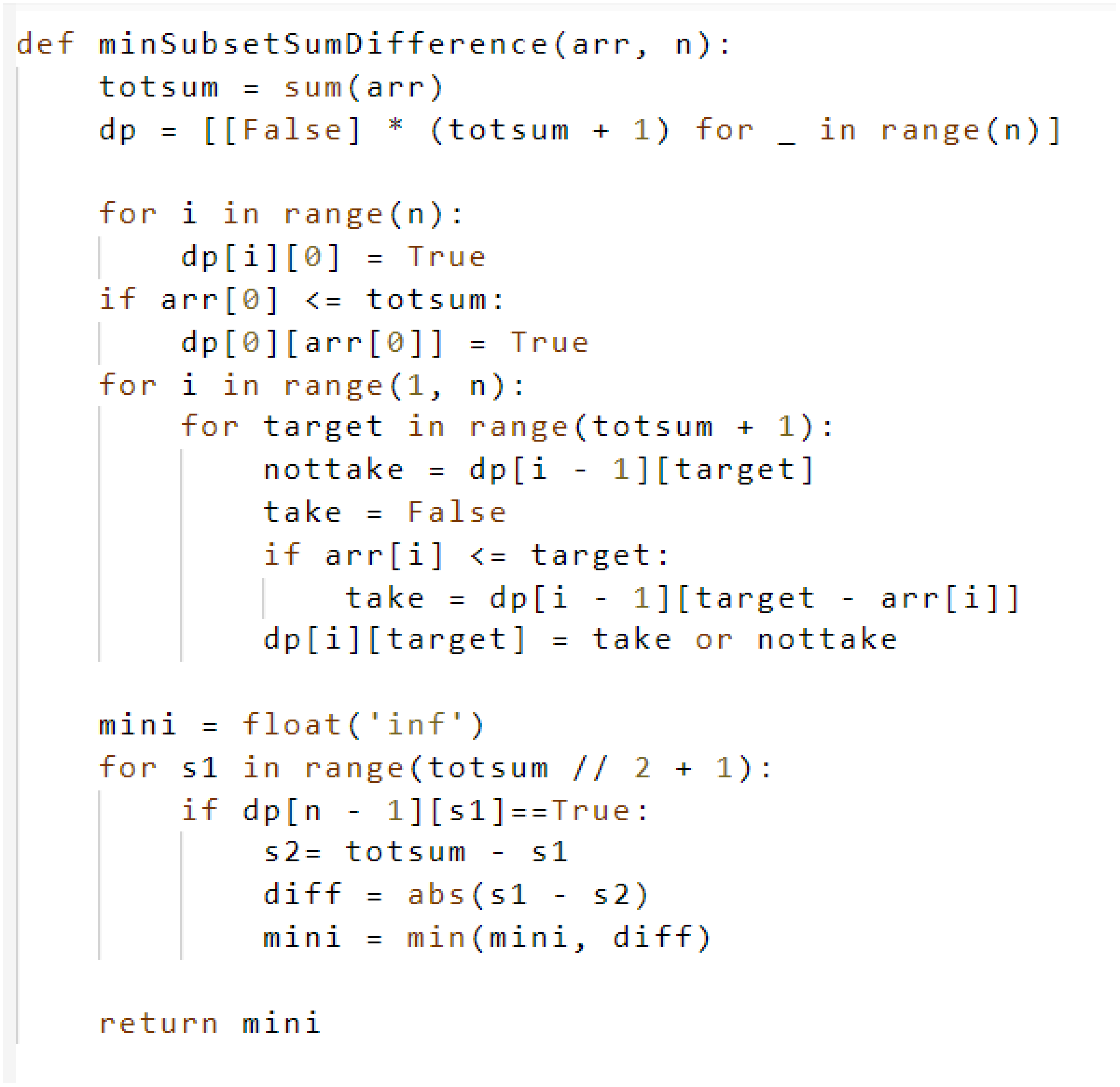
```
class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        n= len(nums)
        s= sum(nums)
        if s%2==0:
            t=s//2
        else:
            return False

        def f(ind,target):
            if target==0:
                return True
            if ind==0:
                if nums[0]==target:
                    return True
                else:
                    return False
            if dp[ind][target]!=-1:
                return dp[ind][target]

            nottake= f(ind-1,target)
            take=False
            if nums[ind]<=target:
                take= f(ind-1,target-nums[ind])

            dp[ind][target]=  take or nottake
            if dp[ind][target]==True:
                return True
            else:
                return False

        dp=[[-1]*(t+1) for i in range(n)]
        return f(n-1,t)
```

Q: Minimum sum difference:

To be discussed, how it works for negative element as well.

Partition a set into two subset so that their absolute difference is minimum when no is positive

Intuition: we use the method of subset sum target to solve it but is works for only positive elemets

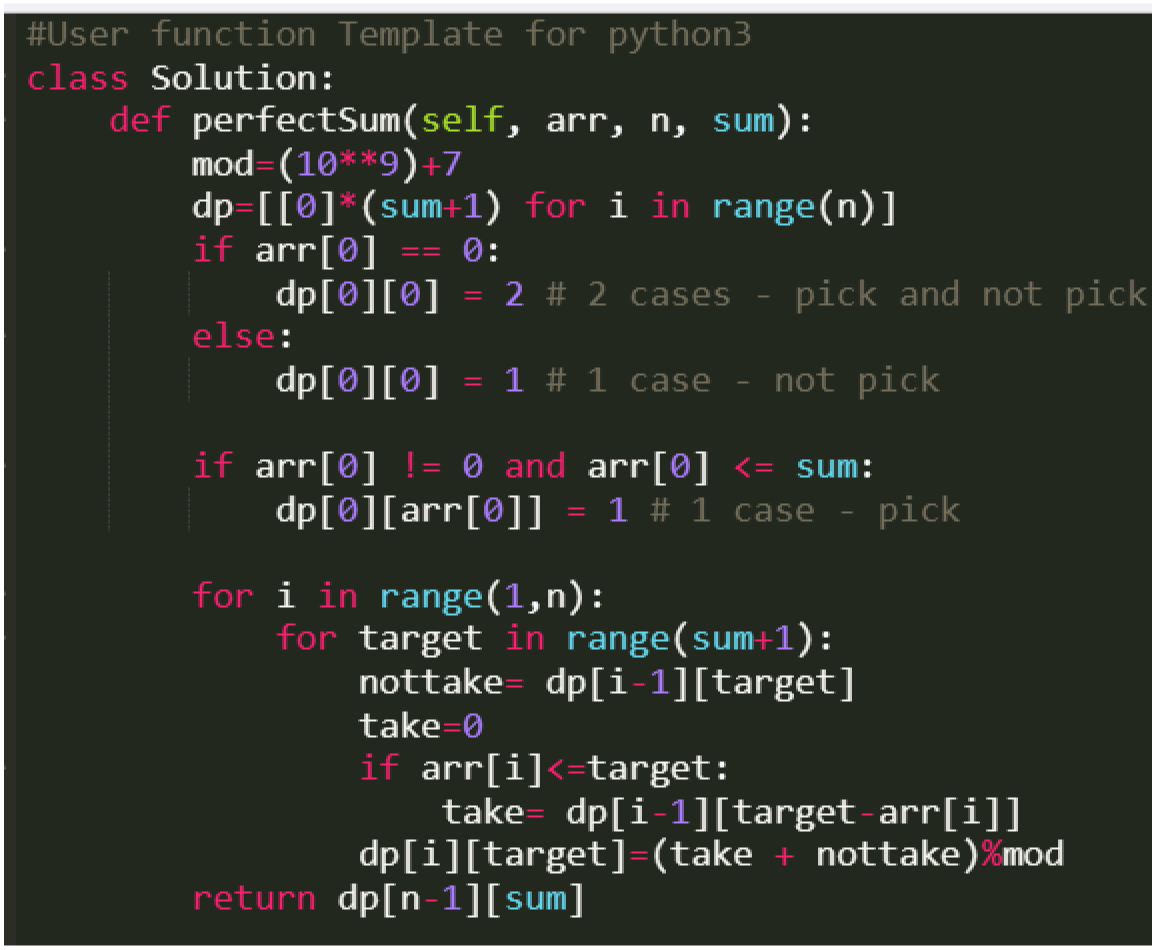


```python
def minSubsetSumDifference(arr, n):
    totsum = sum(arr)
    dp = [[False] * (totsum + 1) for _ in range(n)]

    for i in range(n):
        dp[i][0] = True
    if arr[0] <= totsum:
        dp[0][arr[0]] = True
    for i in range(1, n):
        for target in range(totsum + 1):
            nottake = dp[i - 1][target]
            take = False
            if arr[i] <= target:
                take = dp[i - 1][target - arr[i]]
            dp[i][target] = take or nottake

    mini = float('inf')
    for s1 in range(totsum // 2 + 1):
        if dp[n - 1][s1]==True:
            s2= totsum - s1
            diff = abs(s1 - s2)
            mini = min(mini, diff)

    return mini
```

Q: Count Subsets with Sum K:
Memoization:

```python
#User function Template for python3
class Solution:
    def perfectSum(self, arr, n, sum):
        mod=(10**9)+7
        def f(i,target):
            if i==0:
                if target==0 and arr[0]==0:
                    return 2
                if target==0 or arr[0]==target:
                    return 1
                return 0
            if dp[i][target]!=-1:
                return dp[i][target]
            nottake= f(i-1,target)
            take=0
            if arr[i]<=target:
                take= f(i-1,target-arr[i])
            dp[i][target]=take + nottake
            return (dp[i][target])%mod

        dp=[[-1]*(sum+1) for i in range(n)]
        return f(n-1,sum)
```

Tabulation:



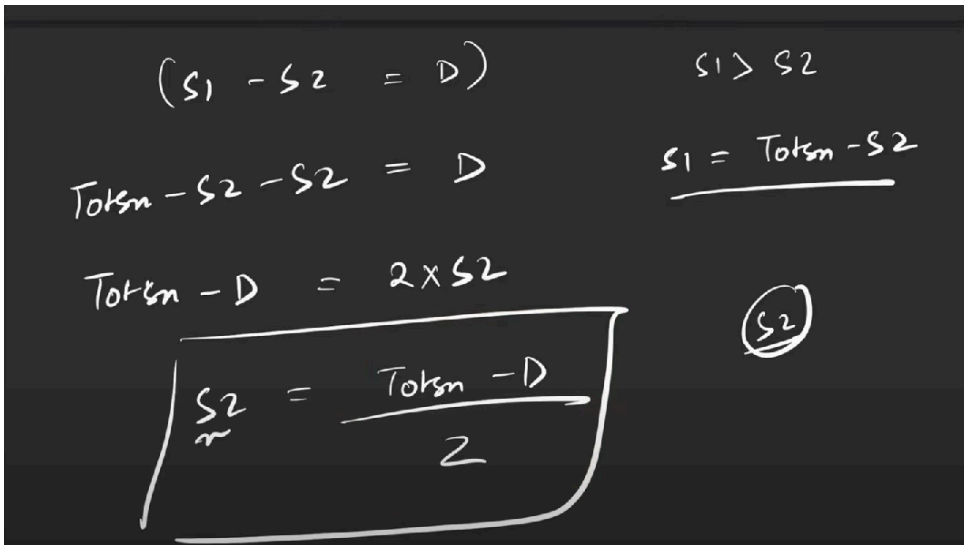
```
#User function Template for python3
class Solution:
    def perfectSum(self, arr, n, sum):
        mod=(10**9)+7
        dp=[[0]*(sum+1) for i in range(n)]
        if arr[0] == 0:
            dp[0][0] = 2 # 2 cases - pick and not pick
        else:
            dp[0][0] = 1 # 1 case - not pick

        if arr[0] != 0 and arr[0] <= sum:
            dp[0][arr[0]] = 1 # 1 case - pick

        for i in range(1,n):
            for target in range(sum+1):
                nottake= dp[i-1][target]
                take=0
                if arr[i]<=target:
                    take= dp[i-1][target-arr[i]]
                dp[i][target]=(take + nottake)%mod
        return dp[n-1][sum]
```

Q: Given an array arr, partition it into two subsets(possibly empty) such that their union is the original array. Let the sum of the element of these two subsets be S1 and S2.
Given a difference d, count the number of partitions in which S1 is greater than or equal to S2 and the difference S1 and S2 is equal to d.

This ques boils down to count the no of subset with target= (totsum-diff)/2

To explain this how this came is :
s1= sum of elements in subset 1
s2 = sum of elements in subset 2
s1 - s2 = d (We need to find 2 subsets with difference d)
s1 + s2 = totalSum (We know the sum of 2 subsets would be equal to the total sum of array)

Adding these 2 equations , we get s1 = (d + totalSum)/2, thus we only need to find a subset with sum s1.
Show less



$(s_1 - s_2 = D)$ $\quad s_1 > s_2$

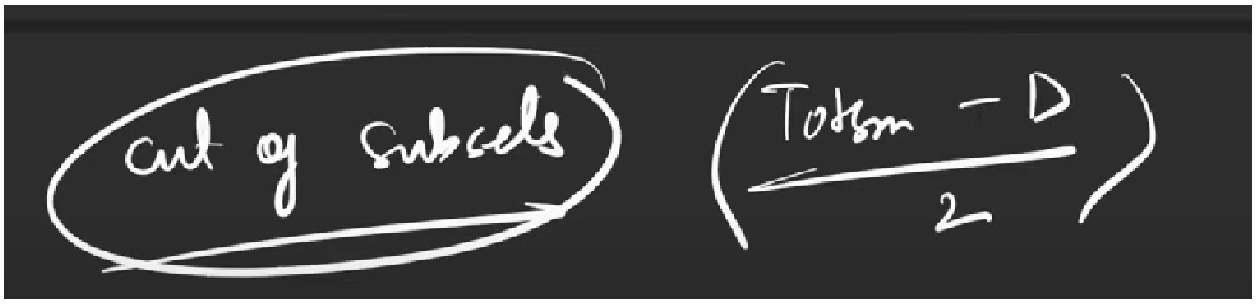
$\underline{s_1 = \text{Totsm} - s_2}$

$\text{Totsm} - s_2 - s_2 = D$

$\text{Totsm} - D = 2 \times s_2$

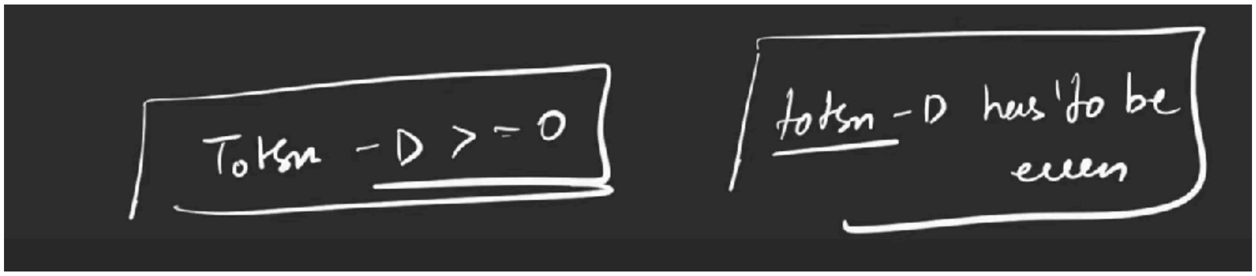
$$\boxed{s_2 = \frac{\text{Totsm} - D}{2}}$$

$\circ s_2$



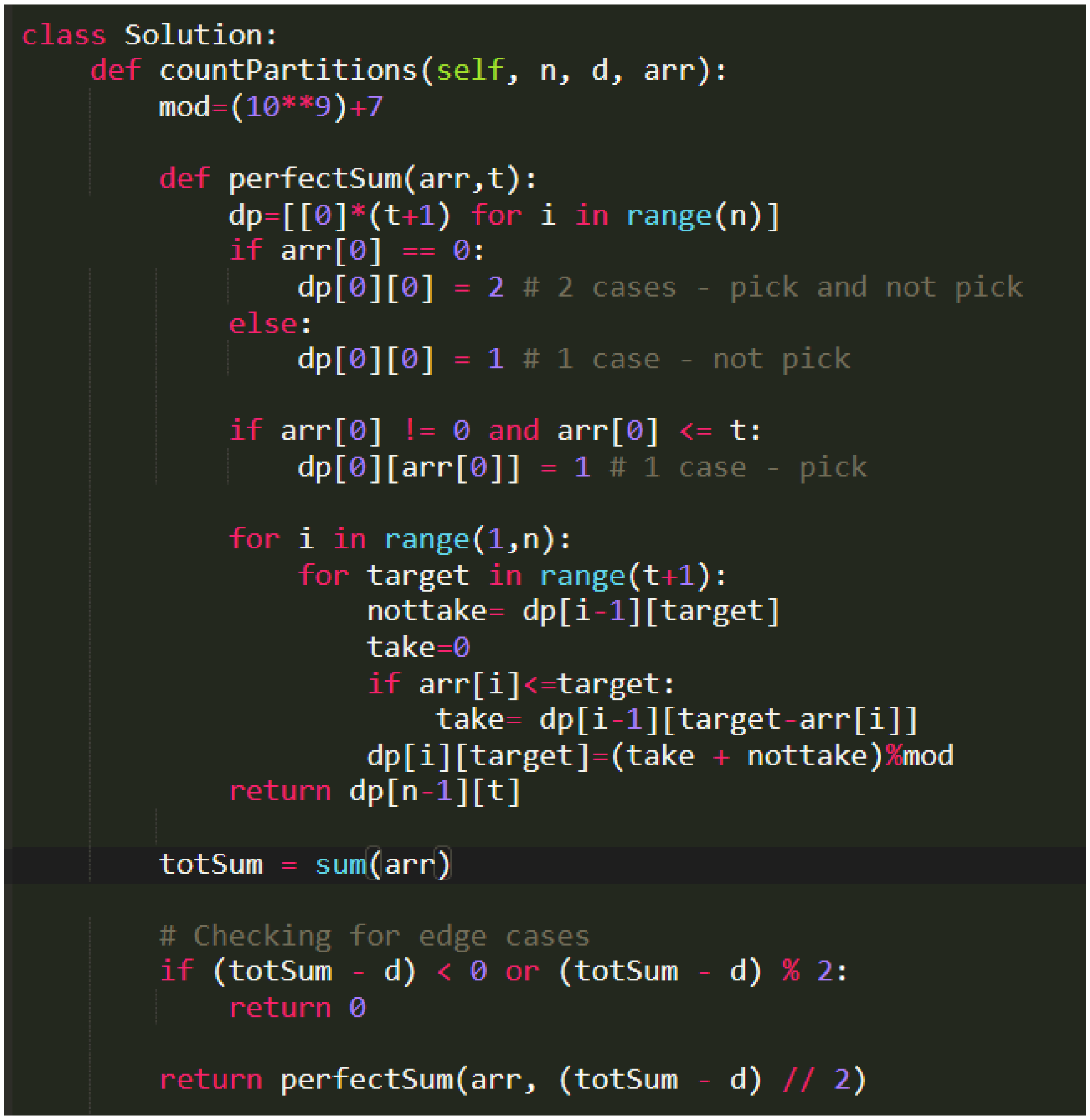
cnt of subsets $\left(\frac{\text{Totsm} - D}{2}\right)$

These two are edge cases:



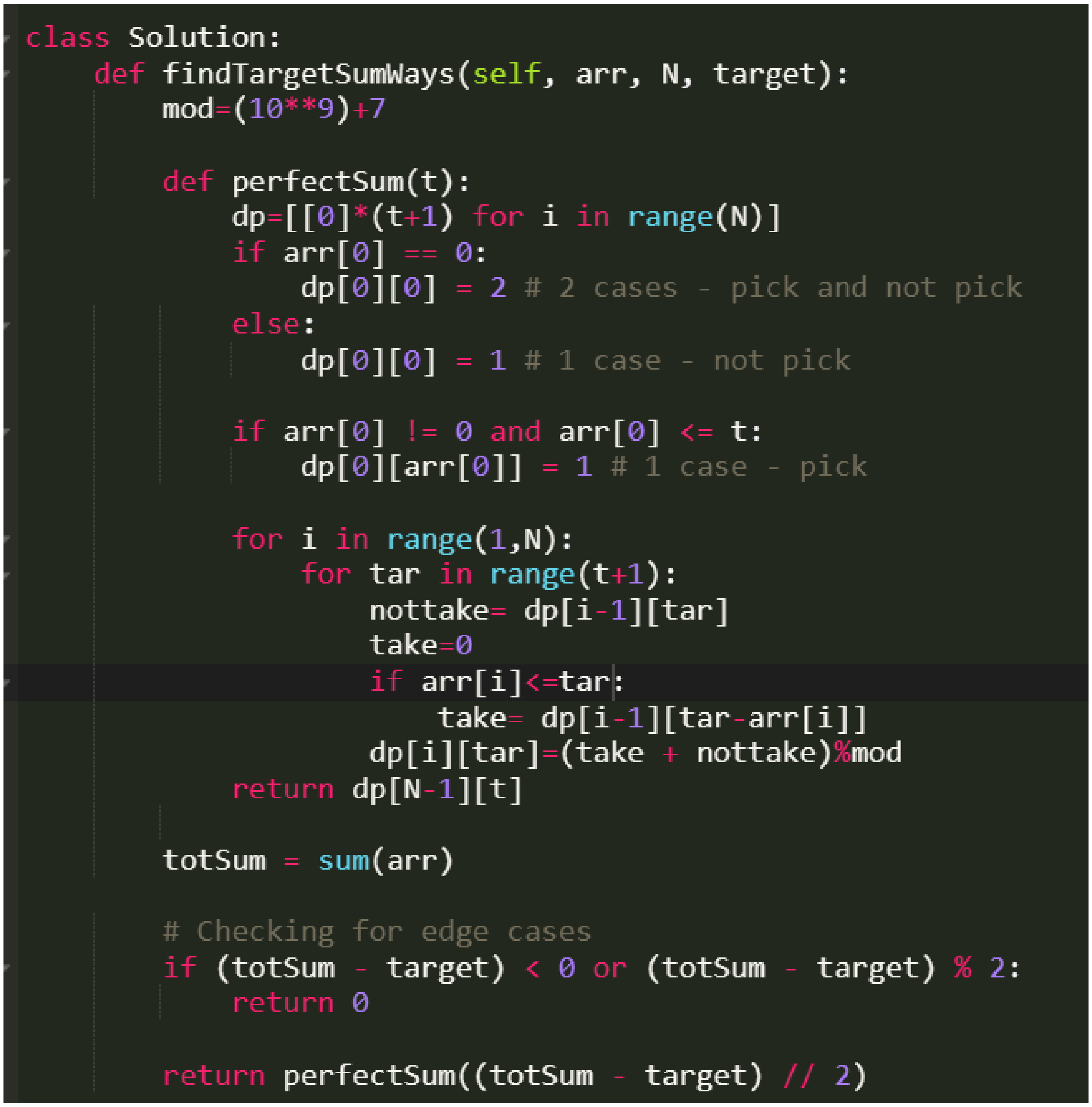
$\boxed{\text{Totsm} - D >= 0}$

$\boxed{\text{totsm} - D \text{ has to be even}}$



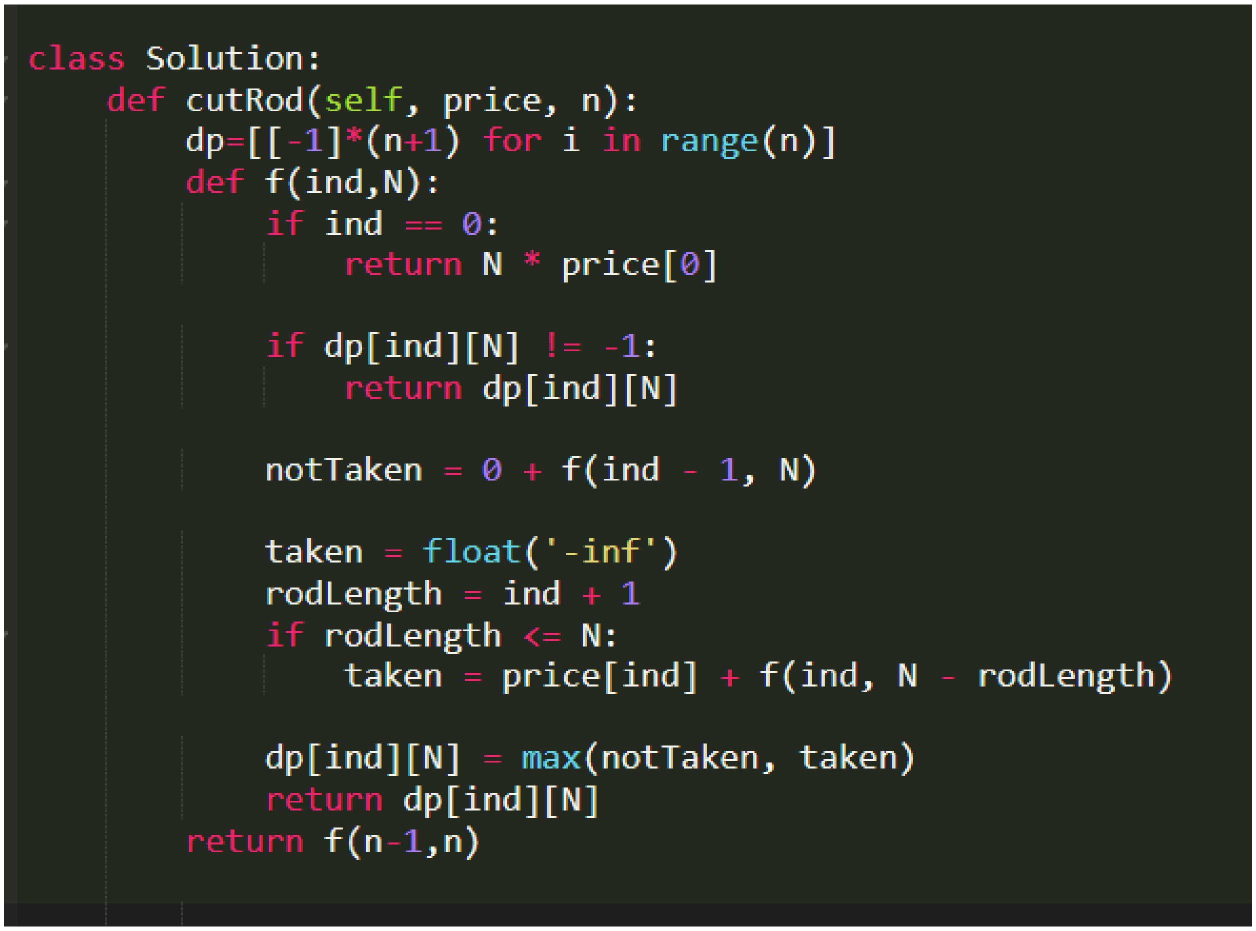
```python
class Solution:
    def countPartitions(self, n, d, arr):
        mod=(10**9)+7

        def perfectSum(arr,t):
            dp=[[0]*(t+1) for i in range(n)]
            if arr[0] == 0:
                dp[0][0] = 2 # 2 cases - pick and not pick
            else:
                dp[0][0] = 1 # 1 case - not pick

            if arr[0] != 0 and arr[0] <= t:
                dp[0][arr[0]] = 1 # 1 case - pick

            for i in range(1,n):
                for target in range(t+1):
                    nottake= dp[i-1][target]
                    take=0
                    if arr[i]<=target:
                        take= dp[i-1][target-arr[i]]
                    dp[i][target]=(take + nottake)%mod
            return dp[n-1][t]

        totSum = sum(arr)

        # Checking for edge cases
        if (totSum - d) < 0 or (totSum - d) % 2:
            return 0

        return perfectSum(arr, (totSum - d) // 2)
```

Q: Subset sum target:
This is similar to divide the set into two subset so their difference is d:

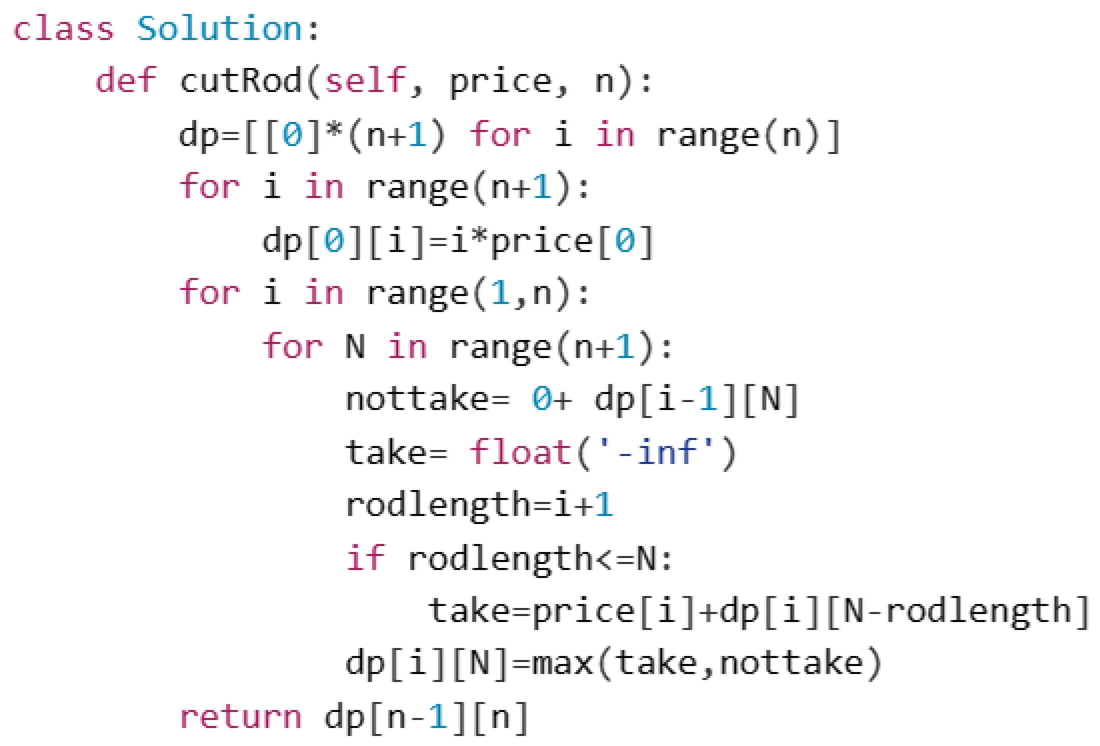


```python
class Solution:
    def findTargetSumWays(self, arr, N, target):
        mod=(10**9)+7

        def perfectSum(t):
            dp=[[0]*(t+1) for i in range(N)]
            if arr[0] == 0:
                dp[0][0] = 2 # 2 cases - pick and not pick
            else:
                dp[0][0] = 1 # 1 case - not pick

            if arr[0] != 0 and arr[0] <= t:
                dp[0][arr[0]] = 1 # 1 case - pick

            for i in range(1,N):
                for tar in range(t+1):
                    nottake= dp[i-1][tar]
                    take=0
                    if arr[i]<=tar:
                        take= dp[i-1][tar-arr[i]]
                    dp[i][tar]=(take + nottake)%mod
            return dp[N-1][t]

        totSum = sum(arr)

        # Checking for edge cases
        if (totSum - target) < 0 or (totSum - target) % 2:
            return 0

        return perfectSum((totSum - target) // 2)
```

Q: rod cutting problem:

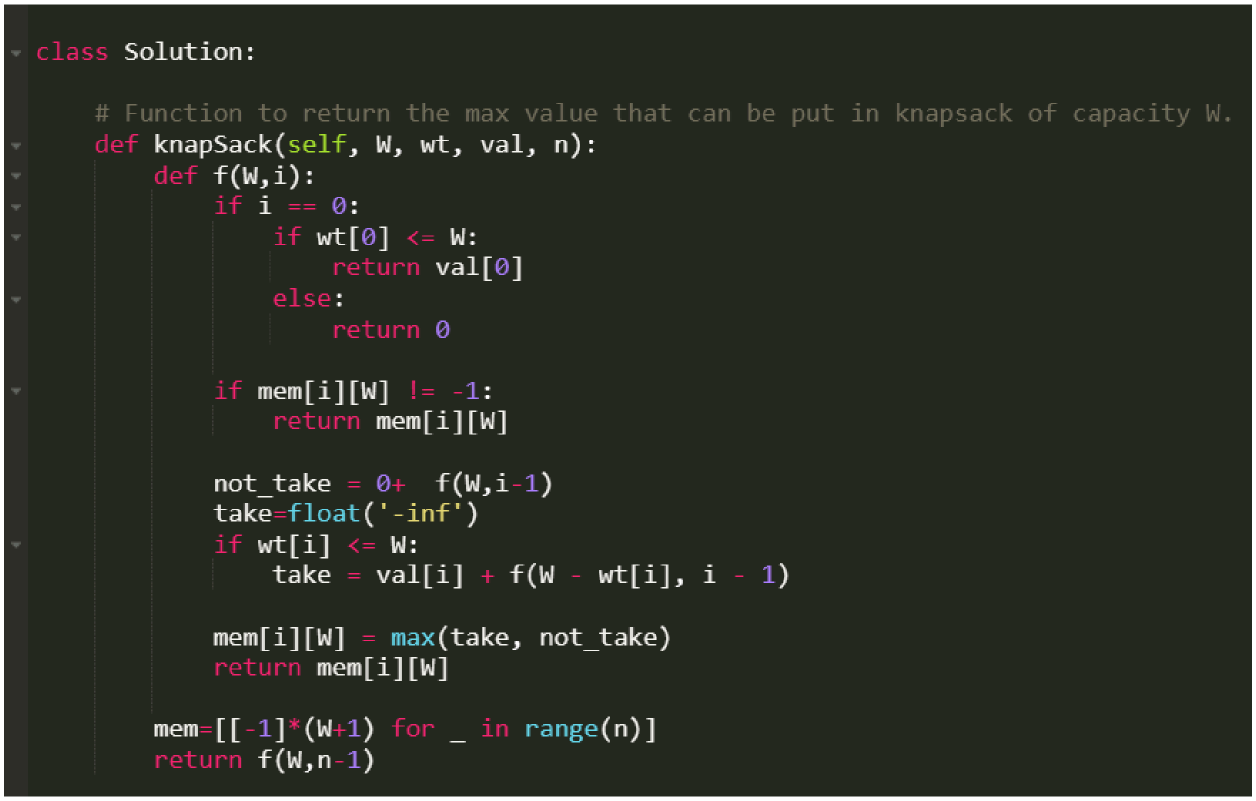


```python
class Solution:
    def cutRod(self, price, n):
        dp=[[-1]*(n+1) for i in range(n)]
        def f(ind,N):
            if ind == 0:
                return N * price[0]

            if dp[ind][N] != -1:
                return dp[ind][N]

            notTaken = 0 + f(ind - 1, N)

            taken = float('-inf')
            rodLength = ind + 1
            if rodLength <= N:
                taken = price[ind] + f(ind, N - rodLength)

            dp[ind][N] = max(notTaken, taken)
            return dp[ind][N]
        return f(n-1,n)
```



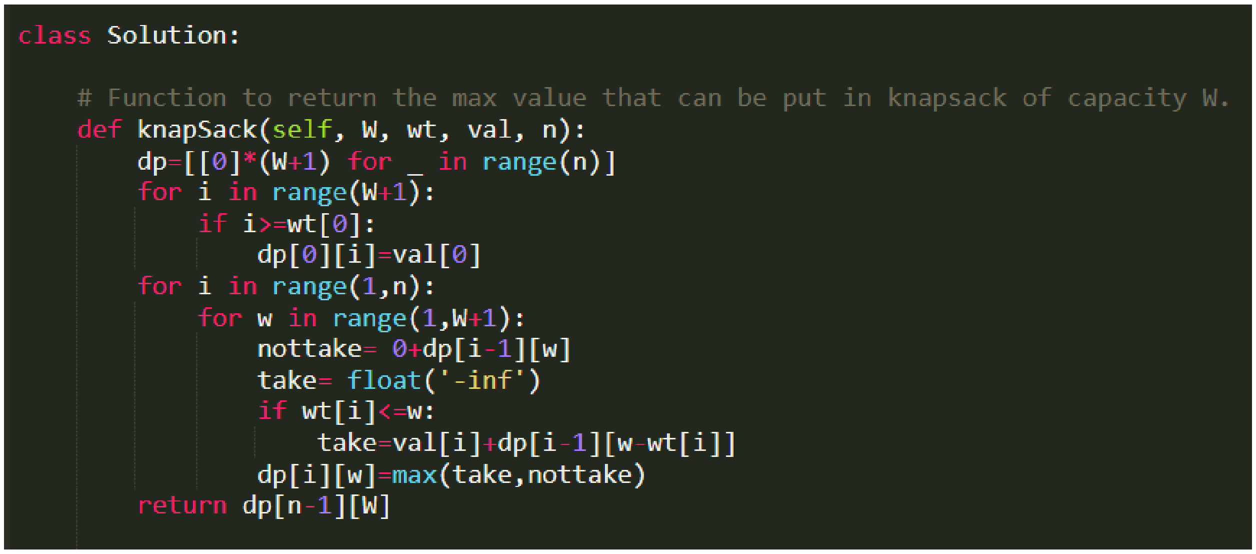
```python
class Solution:
    def cutRod(self, price, n):
        dp=[[0]*(n+1) for i in range(n)]
        for i in range(n+1):
            dp[0][i]=i*price[0]
        for i in range(1,n):
            for N in range(n+1):
                nottake= 0+ dp[i-1][N]
                take= float('-inf')
                rodlength=i+1
                if rodlength<=N:
                    take=price[i]+dp[i][N-rodlength]
                dp[i][N]=max(take,nottake)
        return dp[n-1][n]
```

Q: 0/1 Knapsack:
Memoization:



```python
class Solution:

    # Function to return the max value that can be put in knapsack of capacity W.
    def knapSack(self, W, wt, val, n):
        def f(W,i):
            if i == 0:
                if wt[0] <= W:
                    return val[0]
                else:
                    return 0

            if mem[i][W] != -1:
                return mem[i][W]

            not_take = 0+  f(W,i-1)
            take=float('-inf')
            if wt[i] <= W:
                take = val[i] + f(W - wt[i], i - 1)

            mem[i][W] = max(take, not_take)
            return mem[i][W]

        mem=[[-1]*(W+1) for _ in range(n)]
        return f(W,n-1)
```

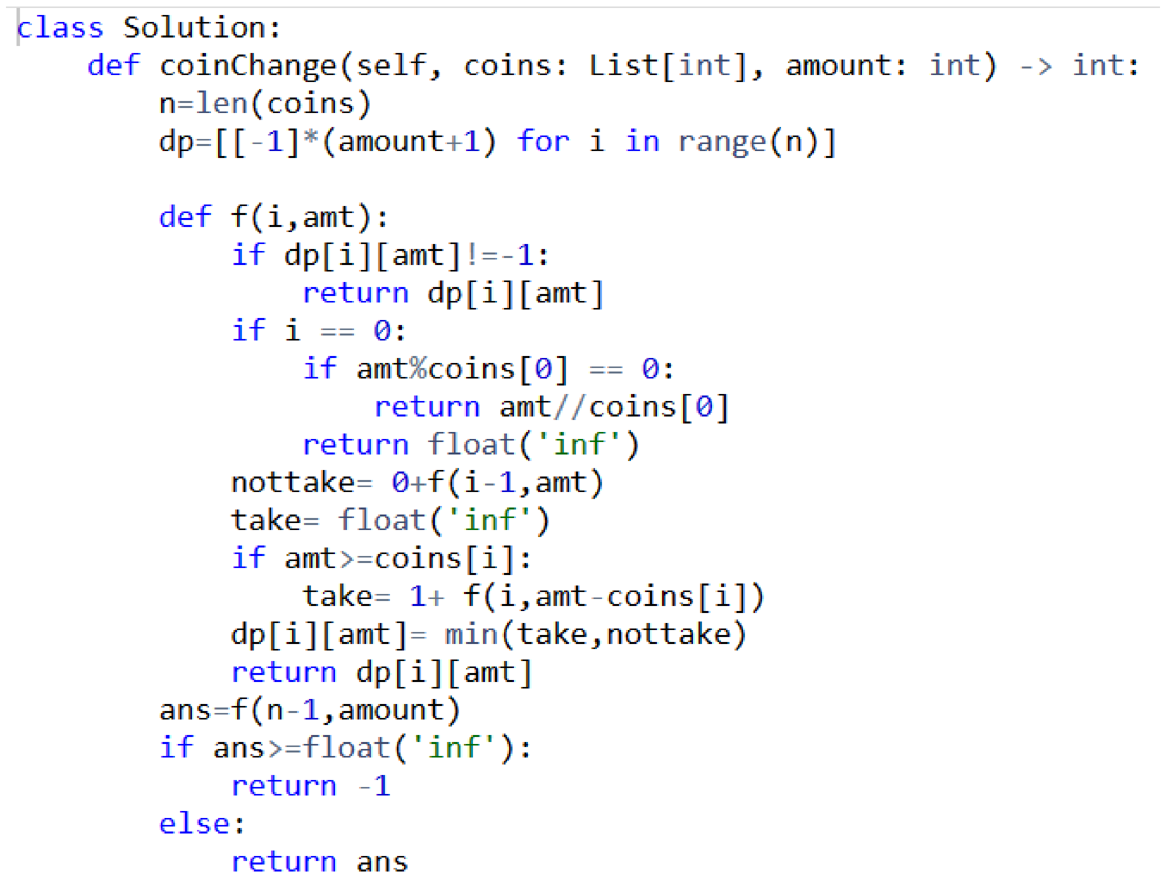
Tabulation:



```python
class Solution:

    # Function to return the max value that can be put in knapsack of capacity W.
    def knapSack(self, W, wt, val, n):
        dp=[[0]*(W+1) for _ in range(n)]
        for i in range(W+1):
            if i>=wt[0]:
                dp[0][i]=val[0]
        for i in range(1,n):
            for w in range(1,W+1):
                nottake= 0+dp[i-1][w]
                take= float('-inf')
                if wt[i]<=w:
                    take=val[i]+dp[i-1][w-wt[i]]
                dp[i][w]=max(take,nottake)
        return dp[n-1][W]
```

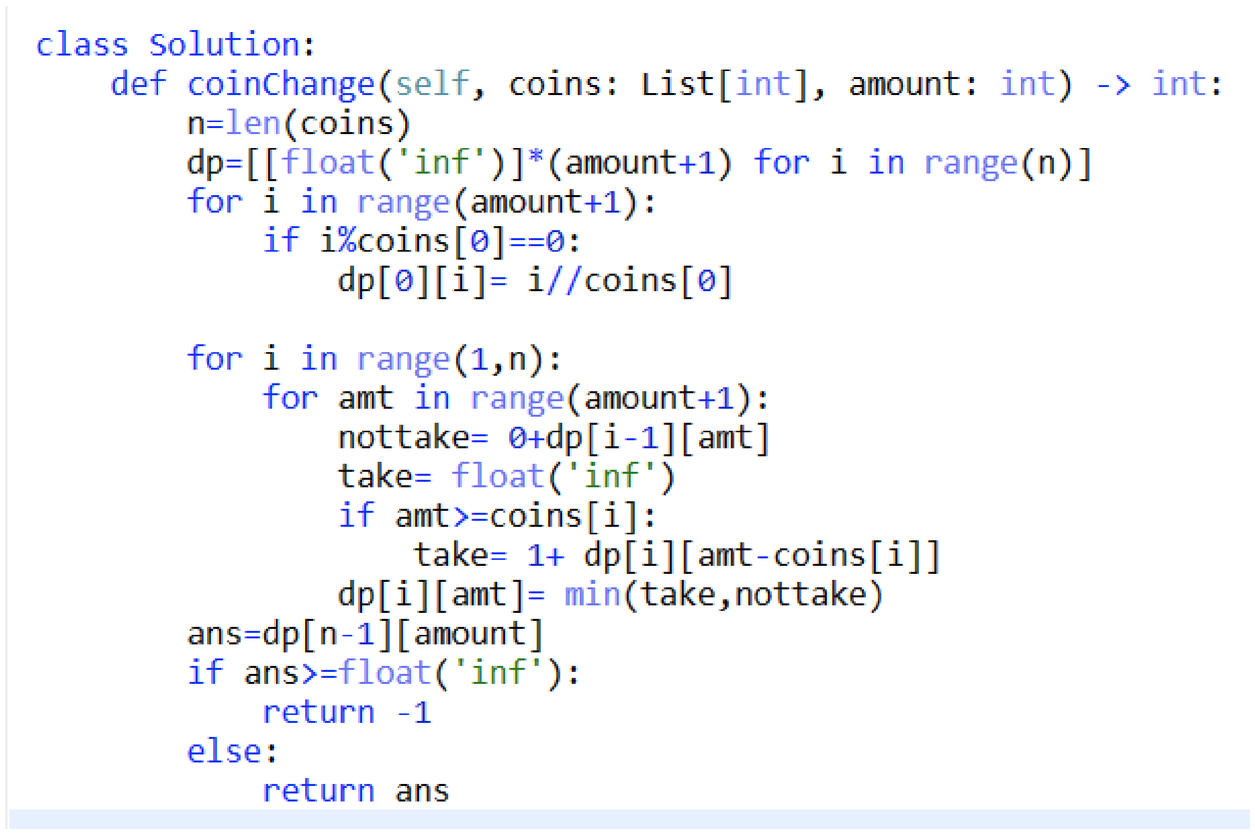
Q: Minimum Coins:

Memoization:



```python
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        n=len(coins)
        dp=[[-1]*(amount+1) for i in range(n)]

        def f(i,amt):
            if dp[i][amt]!=-1:
                return dp[i][amt]
            if i == 0:
                if amt%coins[0] == 0:
                    return amt//coins[0]
                return float('inf')
            nottake= 0+f(i-1,amt)
            take= float('inf')
            if amt>=coins[i]:
                take= 1+ f(i,amt-coins[i])
            dp[i][amt]= min(take,nottake)
            return dp[i][amt]
        ans=f(n-1,amount)
        if ans>=float('inf'):
            return -1
        else:
            return ans
```

Tabulation:

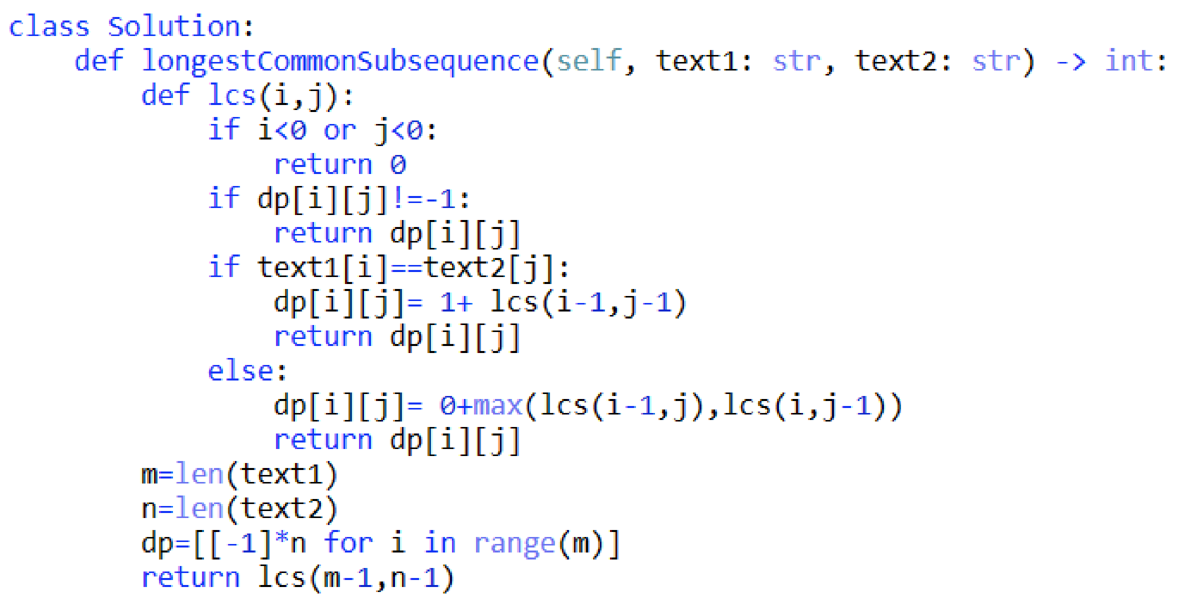


```python
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        n=len(coins)
        dp=[[float('inf')]*(amount+1) for i in range(n)]
        for i in range(amount+1):
            if i%coins[0]==0:
                dp[0][i]= i//coins[0]

        for i in range(1,n):
            for amt in range(amount+1):
                nottake= 0+dp[i-1][amt]
                take= float('inf')
                if amt>=coins[i]:
                    take= 1+ dp[i][amt-coins[i]]
                dp[i][amt]= min(take,nottake)
        ans=dp[n-1][amount]
        if ans>=float('inf'):
            return -1
        else:
            return ans
```

\

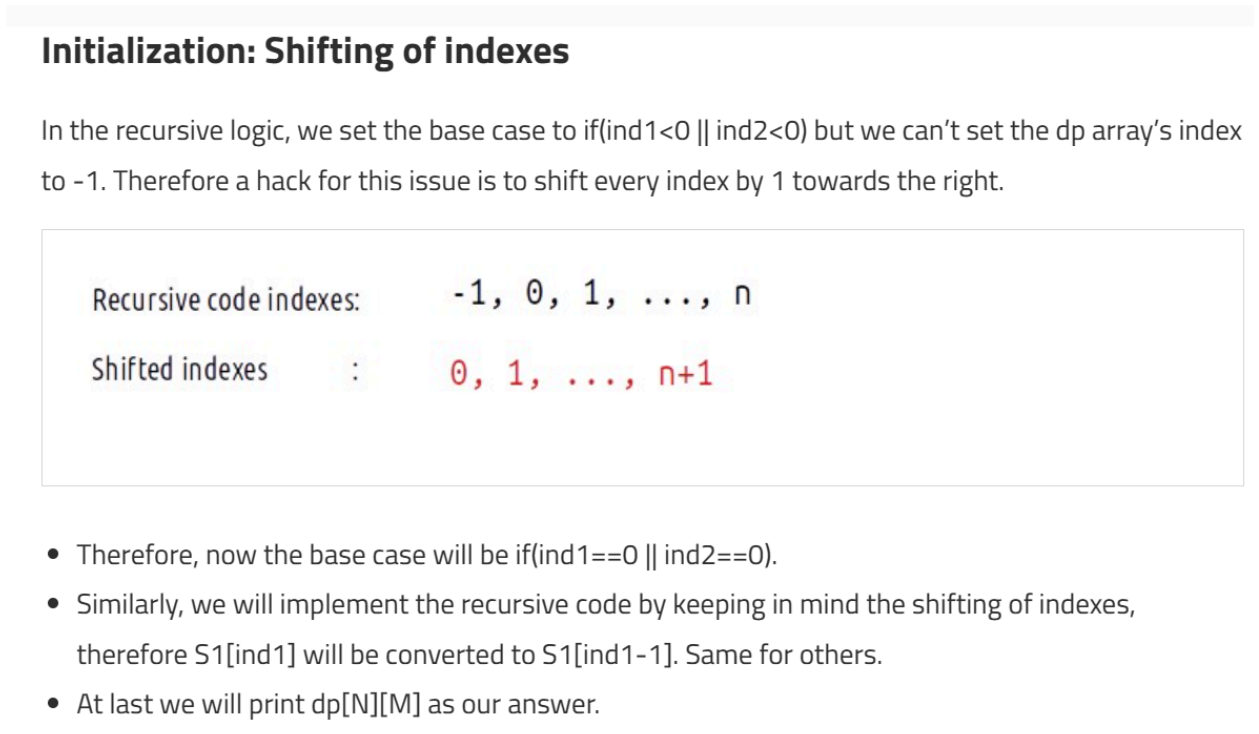
DP ON STRINGS:

Q:Longest common subsequence:



```python
class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        def lcs(i,j):
            if i<0 or j<0:
                return 0
            if dp[i][j]!=-1:
                return dp[i][j]
            if text1[i]==text2[j]:
                dp[i][j]= 1+ lcs(i-1,j-1)
                return dp[i][j]
            else:
                dp[i][j]= 0+max(lcs(i-1,j),lcs(i,j-1))
                return dp[i][j]
        m=len(text1)
        n=len(text2)
        dp=[[-1]*n for i in range(m)]
        return lcs(m-1,n-1)
```

Tabulation: As we see that i<0 and j<0 in tabulation is not possible, so we shift the index.



Initialization: Shifting of indexes

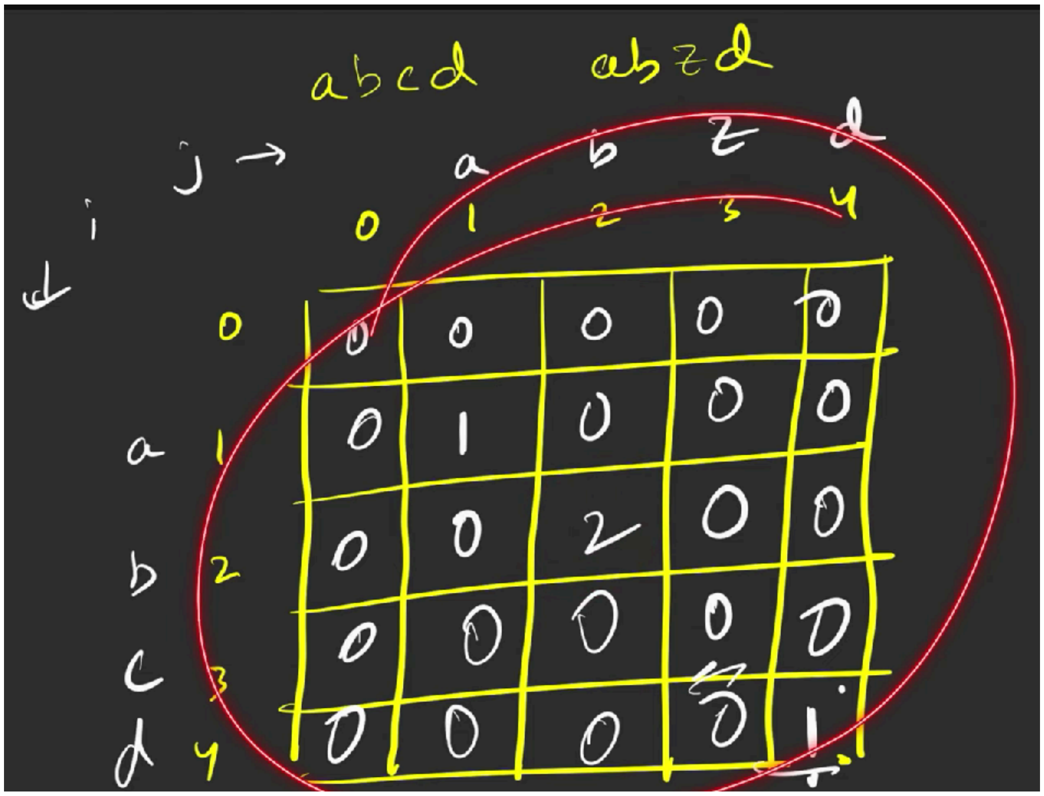
In the recursive logic, we set the base case to if(ind1<0 || ind2<0) but we can't set the dp array's index to -1. Therefore a hack for this issue is to shift every index by 1 towards the right.

```
Recursive code indexes:     -1, 0, 1, ..., n
Shifted indexes        :     0, 1, ..., n+1
```

- Therefore, now the base case will be if(ind1==0 || ind2==0).
- Similarly, we will implement the recursive code by keeping in mind the shifting of indexes, therefore S1[ind1] will be converted to S1[ind1-1]. Same for others.
- At last we will print dp[N][M] as our answer.

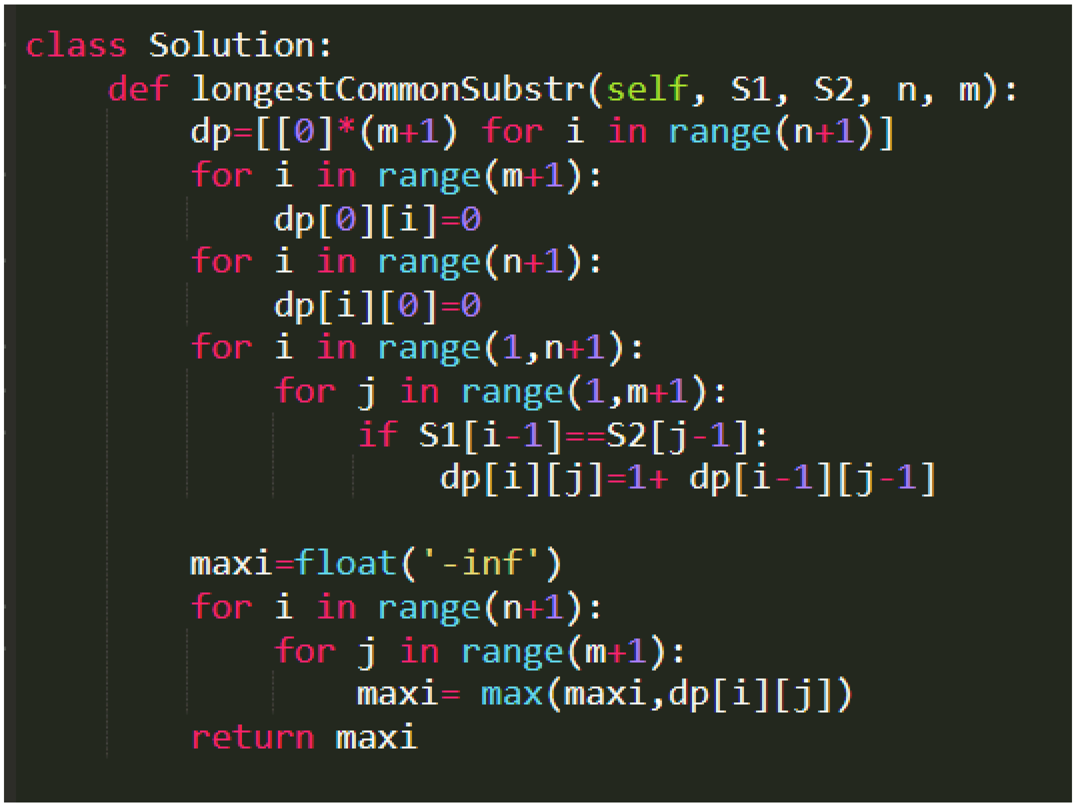
```python
class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        m=len(text1)
        n=len(text2)
        dp=[[0]*(n+1) for i in range(m+1)]
        for i in range(m+1):
            dp[i][0]=0
        for j in range(n+1):
            dp[0][j]=0
        for i in range(1,m+1):
            for j in range(1,n+1):
                if text1[i-1]==text2[j-1]:
                    dp[i][j] = 1+ dp[i-1][j-1]
                else:
                    dp[i][j]= 0+max(dp[i-1][j],dp[i][j-1])
        return dp[m][n]
```

Q: longest common substring:



abcd abzd
j →
i
d

			a	b	z	d
		0	1	2	3	4
	0	0	0	0	0	0
a	1	0	1	0	0	0
b	2	0	0	2	0	0
c	3	0	0	0	0	0
d	4	0	0	0	0	1



```
class Solution:
    def longestCommonSubstr(self, S1, S2, n, m):
        dp=[[0]*(m+1) for i in range(n+1)]
        for i in range(m+1):
            dp[0][i]=0
        for i in range(n+1):
            dp[i][0]=0
        for i in range(1,n+1):
            for j in range(1,m+1):
                if S1[i-1]==S2[j-1]:
                    dp[i][j]=1+ dp[i-1][j-1]

        maxi=float('-inf')
        for i in range(n+1):
            for j in range(m+1):
                maxi= max(maxi,dp[i][j])
        return maxi
```

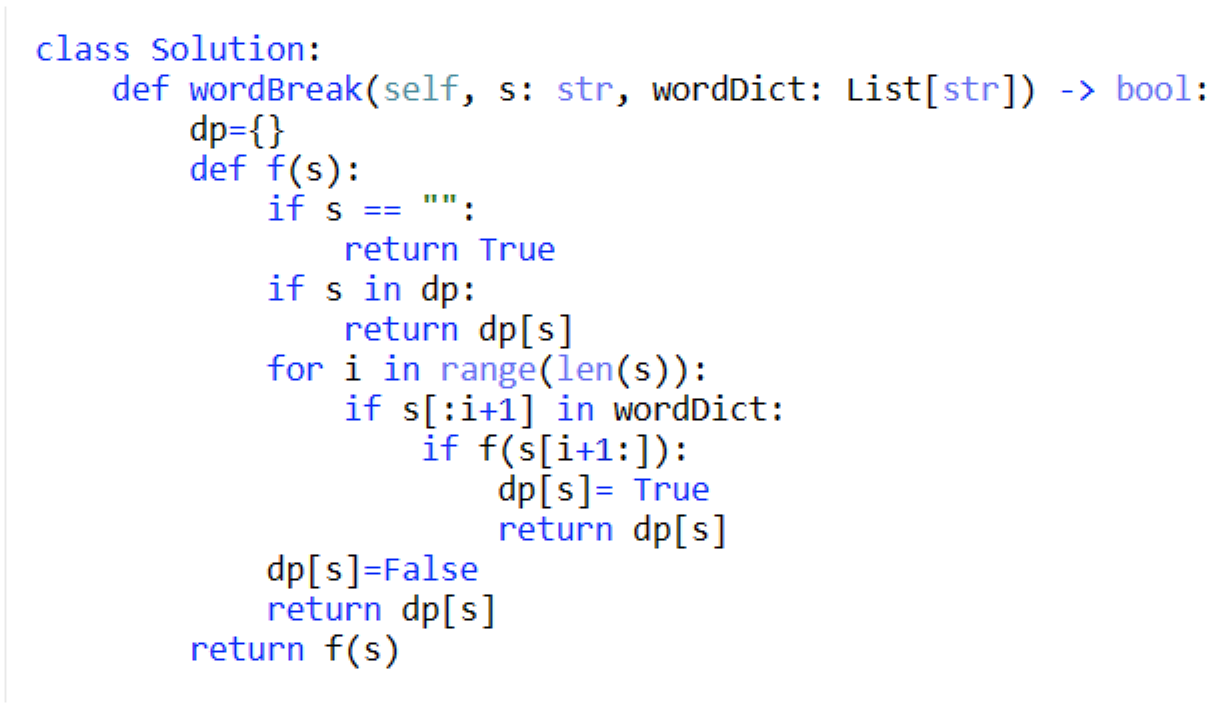
Q:WORD BREAK:

Using Recursion:

Time limit exceeded

```
class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> bool:
        if s == "":
            return True
        for i in range(len(s)):
            if s[:i+1] in wordDict:
                if self.wordBreak(s[i+1:], wordDict):
                    return True
        return False
```

Using DP:MEMOIZATION:



```python
class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> bool:
        dp={}
        def f(s):
            if s == "":
                return True
            if s in dp:
                return dp[s]
            for i in range(len(s)):
                if s[:i+1] in wordDict:
                    if f(s[i+1:]):
                        dp[s]= True
                        return dp[s]
            dp[s]=False
            return dp[s]
        return f(s)
```

DP ON STOCKS:

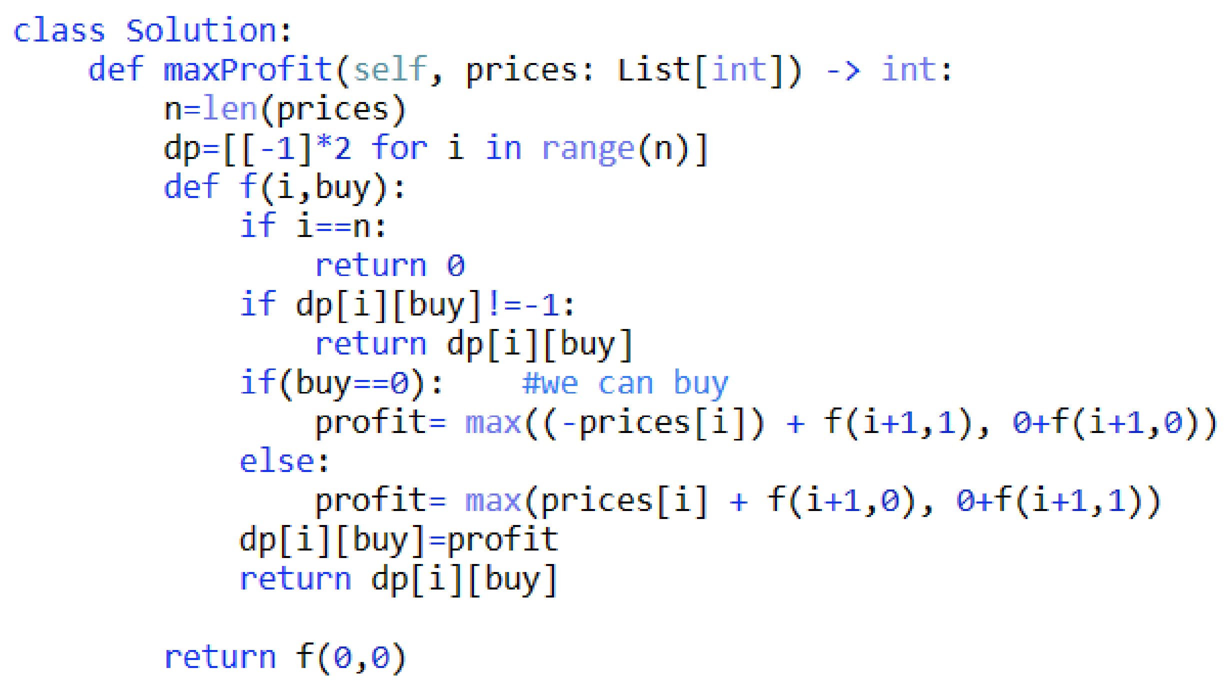
Q1: You are given an integer array `prices` where `prices[i]` is the price of a given stock on the `ith` day.
On each day, you may decide to buy and/or sell the stock. You can only hold at most one share of the stock at any time. However, you can buy it then immediately sell it on the same day.

Find and return *the maximum profit you can achieve*.

Example 1:

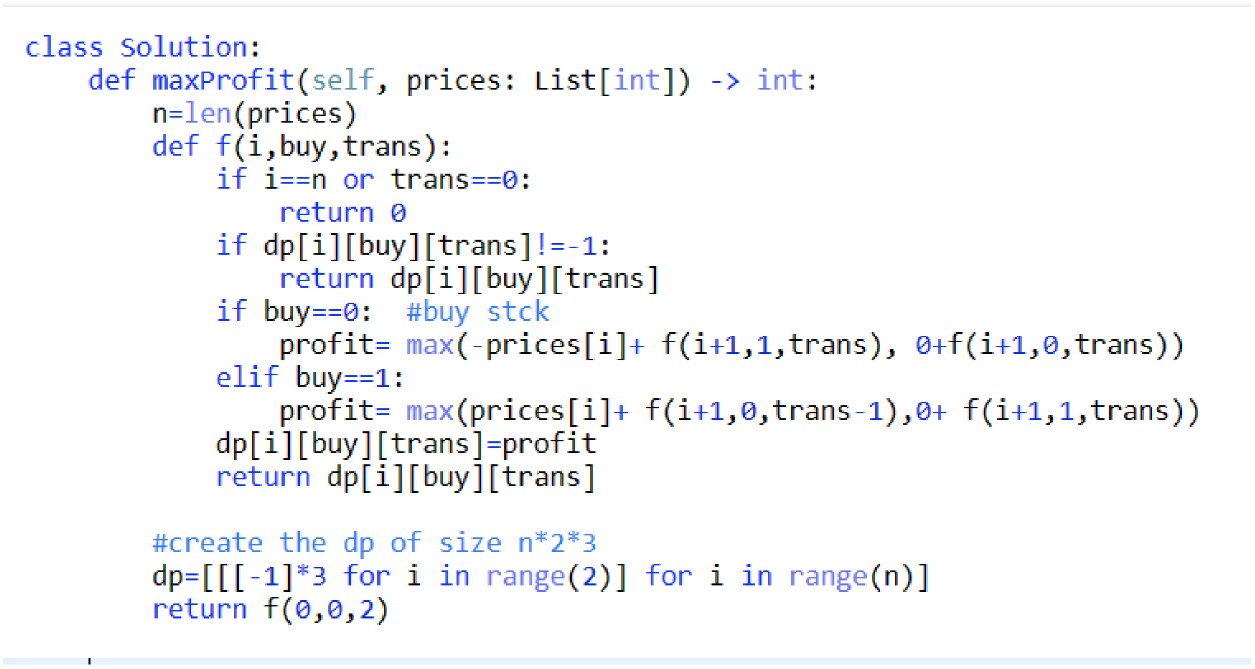
```
Input: prices = [7,1,5,3,6,4]
```

```
Output: 7
Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit =
5-1 = 4.
Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3.
Total profit is 4 + 3 = 7.
```



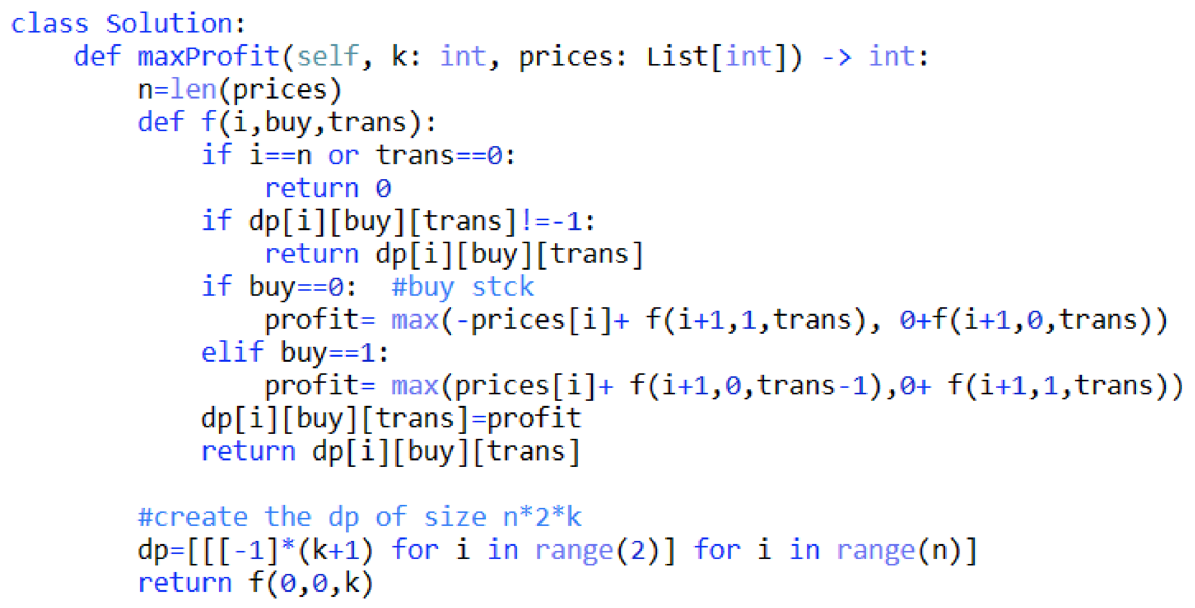
```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        n=len(prices)
        dp=[[-1]*2 for i in range(n)]
        def f(i,buy):
            if i==n:
                return 0
            if dp[i][buy]!=-1:
                return dp[i][buy]
            if(buy==0):    #we can buy
                profit= max((-prices[i]) + f(i+1,1), 0+f(i+1,0))
            else:
                profit= max(prices[i] + f(i+1,0), 0+f(i+1,1))
            dp[i][buy]=profit
            return dp[i][buy]

        return f(0,0)
```

Q2:Dp on stocks: when given that no of transaction allowed is 2 only:



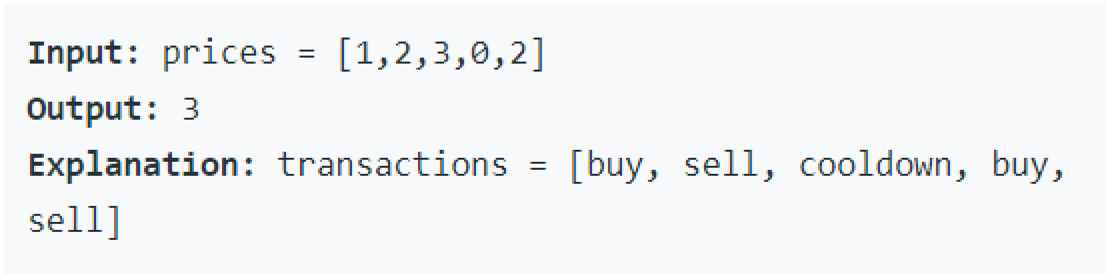
```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        n=len(prices)
        def f(i,buy,trans):
            if i==n or trans==0:
                return 0
            if dp[i][buy][trans]!=-1:
                return dp[i][buy][trans]
            if buy==0:  #buy stck
                profit= max(-prices[i]+ f(i+1,1,trans), 0+f(i+1,0,trans))
            elif buy==1:
                profit= max(prices[i]+ f(i+1,0,trans-1),0+ f(i+1,1,trans))
            dp[i][buy][trans]=profit
            return dp[i][buy][trans]

        #create the dp of size n*2*3
        dp=[[[-1]*3 for i in range(2)] for i in range(n)]
        return f(0,0,2)
```

Q3: Dp on stocks when no of transactions allowed is atmost k:

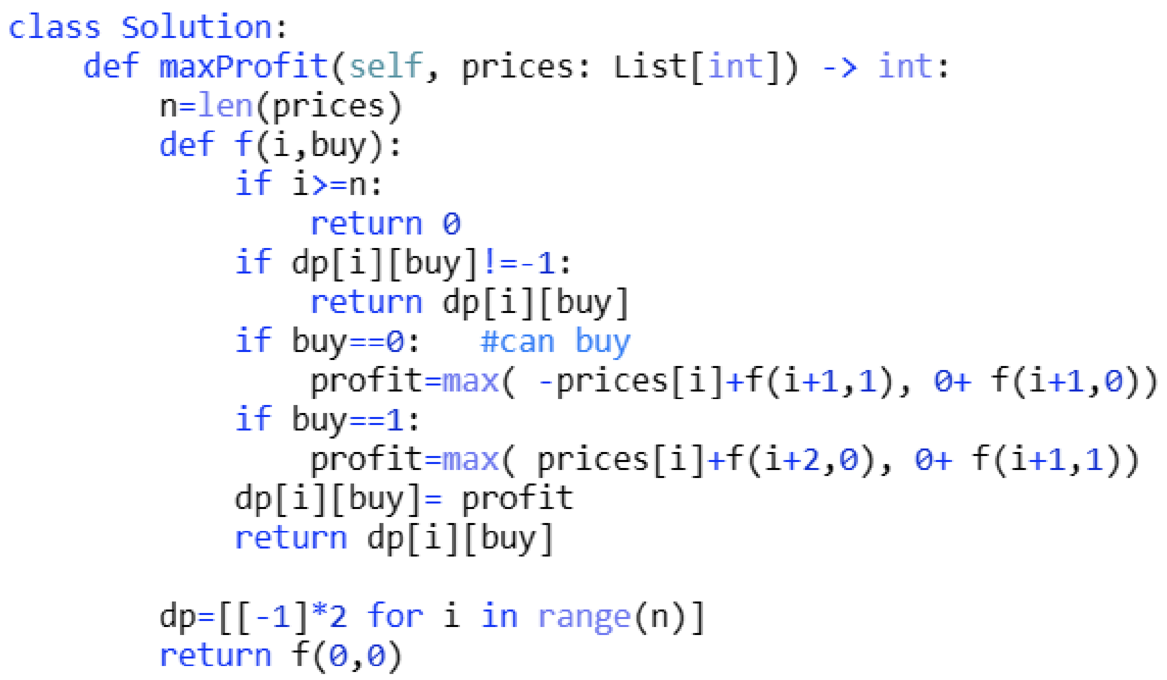


```
class Solution:
    def maxProfit(self, k: int, prices: List[int]) -> int:
        n=len(prices)
        def f(i,buy,trans):
            if i==n or trans==0:
                return 0
            if dp[i][buy][trans]!=-1:
                return dp[i][buy][trans]
            if buy==0:  #buy stck
                profit= max(-prices[i]+ f(i+1,1,trans), 0+f(i+1,0,trans))
            elif buy==1:
                profit= max(prices[i]+ f(i+1,0,trans-1),0+ f(i+1,1,trans))
            dp[i][buy][trans]=profit
            return dp[i][buy][trans]

        #create the dp of size n*2*k
        dp=[[[-1]*(k+1) for i in range(2)] for i in range(n)]
        return f(0,0,k)
```

Q4: Dp on stock with cooldown:
Cooldown means after one transaction we cannot do next transaction after day. We have to wait for one day after completion of transaction to do other transactions.



```
Input: prices = [1,2,3,0,2]
Output: 3
Explanation: transactions = [buy, sell, cooldown, buy,
sell]
```



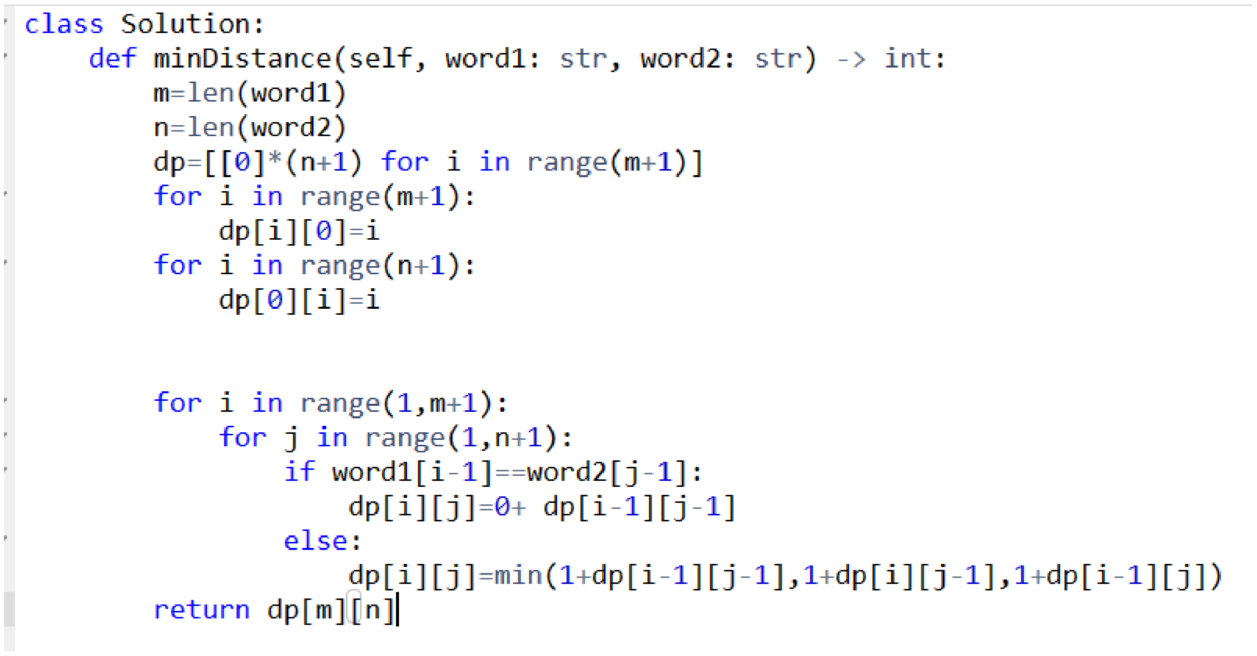
```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        n=len(prices)
        def f(i,buy):
            if i>=n:
                return 0
            if dp[i][buy]!=-1:
                return dp[i][buy]
            if buy==0:   #can buy
                profit=max( -prices[i]+f(i+1,1), 0+ f(i+1,0))
            if buy==1:
                profit=max( prices[i]+f(i+2,0), 0+ f(i+1,1))
            dp[i][buy]= profit
            return dp[i][buy]

        dp=[[-1]*2 for i in range(n)]
        return f(0,0)
```

Q: DP on stocks with transaction fee:
In this question after finishing of one complete transaction(buy and sell) fee is charged.

```python
class Solution:
    def maxProfit(self, prices: List[int], fee: int) -> int:
        n=len(prices)
        def f(i,buy):
            if i>=n:
                return 0
            if dp[i][buy]!=-1:
                return dp[i][buy]
            if buy==0:
                profit= max(-prices[i]+f(i+1,1), 0+ f(i+1,0))
            if buy==1:
                profit= max(prices[i]+f(i+1,0)-fee, 0+ f(i+1,1))
            dp[i][buy]=profit
            return dp[i][buy]

        dp= [[-1]*2 for i in range(n)]
        return f(0,0)
```

DP: Edit distance:

```python
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        m=len(word1)
        n=len(word2)
        dp=[[-1]*(n) for i in range(m)]
        def f(i,j):
            if i<0:
                return j+1
            if j<0:
                return i+1
            if dp[i][j]!=-1:
                return dp[i][j]
            if word1[i]==word2[j]:
                dp[i][j]= 0+f(i-1,j-1)
            else:
                dp[i][j]= 1+ min(f(i-1,j-1),f(i-1,j),f(i,j-1))
            return dp[i][j]
        return f(m-1,n-1)
```



```python
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        m=len(word1)
        n=len(word2)
        dp=[[0]*(n+1) for i in range(m+1)]
        for i in range(m+1):
            dp[i][0]=i
        for i in range(n+1):
            dp[0][i]=i


        for i in range(1,m+1):
            for j in range(1,n+1):
                if word1[i-1]==word2[j-1]:
                    dp[i][j]=0+ dp[i-1][j-1]
                else:
                    dp[i][j]=min(1+dp[i-1][j-1],1+dp[i][j-1],1+dp[i-1][j])
        return dp[m][n]
```