PARTITION METHOD

Question : 131. Palindrome Partitioning

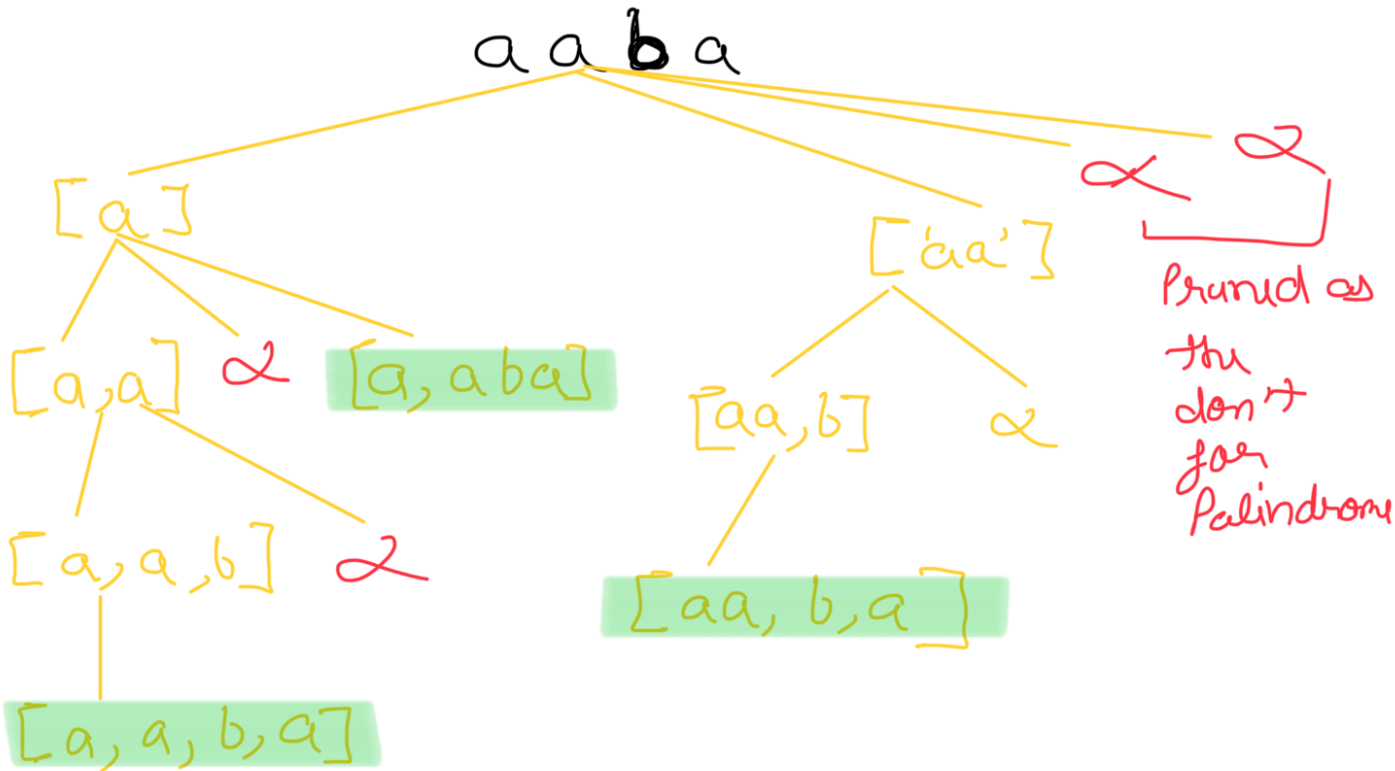
Input : aaba

Output : [['a','a','b','a'], ['aa','b','a'], ['a','aba']]

Explanation : a|a|b|a

aa|b|a

a|aba



CODE :

```
def partition(s):
    n = len(s)
    res, curr = [], []
    dp = [[-1]*n for _ in range(n)]
    for l in range(1, n+1):
        for i in range(n-l+1):
            j = i + l - 1
            if i == j:
                dp[i][j] = True
            elif s[i] == s[j] and (j == i+1
                                   or (dp[i+1][j-1])
                dp[i][j] = True
            else:
                dp[i][j] = False
    def backtrack(index):
        if index > n-1:
            res.append(curr[:])
            return
        for i in range(n):
            if dp[index][i]:
                curr.append(s[index:i+1])
                backtrack(i+1)
                curr.pop()
```

```
backtrack[0]
return res
```

132. Palindrome Partitioning II

Input : s = "aab"

Output = 1

Explanation : we need to find minimum cuts, in this case [aa,b], 1 cut is needed

RECURSIVE APPROACH

```
def minCut(s):
    n = len(s)
    def isPalindrome(i, j):
        while i > j:
            if s[i] != s[j]:
                return False
            i += 1
            j -= 1
        return True
    def partition(start, index):
```

```
# base, cases
if start == end or isPalindrome(start, index):
    return 0

minimum = start - end  # maximum cut required

for j in range(start, end):
    if is_Palindrome(start, j):
        minimum = min(minimum, partition(j+1, end))

return minimum
```

* MEMOIZATION APPROACH

```
def minCut(s):

    n = len(s)
    isPalindrome = [[None]*n for _ in range(n)]
    minCuts = [[None]*n for _ in range(n)]

    for l in range(1, n+1)
        for i in range(n-l+1)

            j = n+l-1
            if i == j:
                isPalindrome[i][j] = True
            elif s[i] == s[j] and (j == i+1 or
                                   isPalindrome[i+1][j-1]):
```

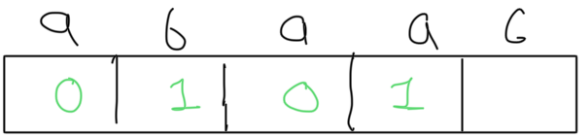
```
                isPalindrome[i][j] = True
            else:
                isPalindrome[i][j] = False
    def partitions(start, end):
        if start == end or isPalindrome(start, end):
            return 0
        if minCuts[start][end] is not None:
            return minCuts[start][end]
        minimum = end - start
        for endindex in range(start, end)
            if isPalindrome[start][endindex]:
                minimum = min(minimum, 1 + partitions(
                                endindex + 1, end ))
        mincut[start][end] = minimum
        return mincut[start][end]
    return partitions[0][len(s)-1]
```

TABULATION APPROACH

```
1  class Solution:
2      def minCut(self, s: str) -> int:
3          n = len(s)
4          dp = [[0]*n for _ in range(n)]
5
6          for l in range(1,n+1):
7              for i in range(n-l+1):
8                  j = i + l - 1
```

```
                # check for single digit like "a", so 0 cuts are required
                if i == j:
                    dp[i][j] = 0
                # check for like "aa" or "aaa" which are already palindromes
                elif s[i] == s[j] and (j == i + 1 or dp[i+1][j-1] == 0):
                    dp[i][j] = 0
                # check for non palindromic characters like "abac", by iterating with "k"
                else:
                    dp[i][j] = j - i #maximum cuts i.e., a|b|a|c
                    for k in range(i,j):
                        dp[i][j] = min(dp[i][j],dp[i][k] + 1 + dp[k+1][j])
                        #check for 1 to n cuts i.e., a|bac,ab|ac,aba|c
        return dp[0][n-1]
```

TABULATION APPROACH - 1D



a|b|a|a|c = 4

a b a a c

1 + cuts [a b a a] = 2

min(4, 2) = 2

```
class Solution:
    def minCut(self, s: str) -> int:
        n = len(s)
        isPalindrome = [[False]*n for _ in range(n)]
        dp = [0]*n

        for l in range(1,n+1):
            for i in range(n -l + 1):
                j = i + l - 1
                if i == j:
                    isPalindrome[i][j] = True
                elif s[i] == s[j] and (j == i + 1 or isPalindrome[i+1][j-1]):
                    isPalindrome[i][j] = True
```

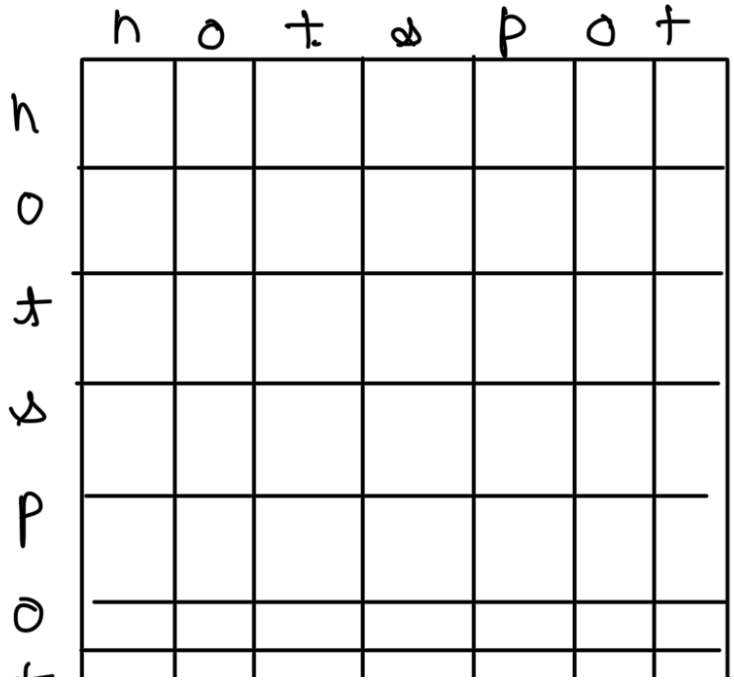
```
                isPalindrome[i][j] = True
            else:
                isPalindrome[i][j] = False

    for end in range(n):
        min_cuts = end
        for start in range(end+1):
            if isPalindrome[start][end]:
                if start == 0:
                    min_cuts = 0
                else :
                    min_cuts = min(min_cuts, 1 + dp[start-1])
        dp[end] = min_cuts

    return dp[n-1]
```

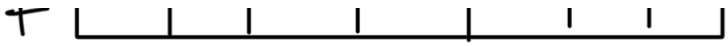
139. Word Break

Input : s = "leetcode", wordDict = ["leet", "code"]

Output = True

Tabulation Approach 1





CODE

```
def wordbreak(s, worddict):

    n = len(s)
    dp = [ [-1]*n for n in range(n)

    for l in range(1, n+1)
        for i in rang(n-l+1)

            j = i+l-1

            if s[i:j+1] in wordDict:
                dp[i][j] = True

            else:
                for k in range(i,j):
                    dp[i][j] = dp[i][j] or

                    (dp[i][k] and dp[k+1][j])

    return [0][n-1]
```

2nd APPROACH : MEMOIZATION

s = "hotspot"

wordDict = ["hot", "spot"]

0	1	2	3	4	5	6
h	o	t	s	p	o	t

1) need a word from wordDict that ends at this index

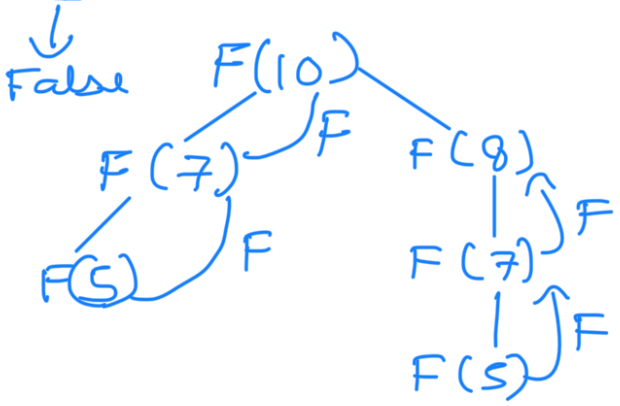
for every word check if → s[i − len(word) to i] is +1 in word dict

function(index i)
Recursive

0 1 2 3 4 5 6 7 8 9 10

s = "b b b b b a b b b b b", worddict = ['b', 'bb', 'bbb']

False



CODE :

```
def wordBreak(s, wordDict):

    n = len(s)
    dp = [-1]*n

    def check(index):

      if i<0 :
        return 0

      if dp[i] != -1 :
```

```
        return dp[i]
    for word in wordDict:
        if s[i-len(word)+1 : i] == word and
                                check(i - len(word)):
            dp[i] = True
            return dp[i]
    dp[i] = False
    return dp[i]
return check(n-1)
```

TABULATION APPROACH :

```
def wordBreak(s, wordDict):
    n = len(s)
    dp = [False] * n
    for i in range(n)
        for word in wordDict:
            if i < len(word) - 1 :
                continue
            elif s[ i - len(word) + 1 : i+1 ] ==
                word and (i == len(word)-1 or
                                dp[i - len(word)])
                dp[i] = True
```

break

return dp[n-1]

QUESTION : MATRIX CHAIN MULTIPLICATION

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The efficient way is the one that involves the least number of multiplications.

The dimensions of the matrices are given in an array **arr[]** of size **N** (such that N = number of matrices + 1) where the **i^{th}** matrix has the dimensions **(arr[i-1] x arr[i])**.

Example 1:

Input: N = 5
arr = {40, 20, 30, 10, 30}
Output: 26000
Explanation: There are 4 matrices of dimension 40x20, 20x30, 30x10, 10x30. Say the matrices are named as A, B, C, D. Out of all possible combinations, the most efficient way is (A*(B*C))*D.
The number of operations are -
20*30*10 + 40*20*10 + 40*10*30 = 26000.

Observations:

If we are multiplying two matrices of dimension [2X3] and [3X1] then

total no. of multiplication = 2X3X1

$$R_1 \times C_1 / R_2 \times C_2$$

Ex : arr = [20, 10, 5, 30] 20X10, 10X5, 5X30

Two way of multiplying

1 → $(20^{1}X10\ \ 10^{2}X5)(5^{3}X30)$ = 20X10X5 + 20X5X30
20X5
= 4000

2 → $(20^{1}X10)(10^{2}X5\ \ 5^{3}X30)$ = 20X10X30 + 10X5X30
10X30
= 7500

So '1' is the most efficient way of multiplying

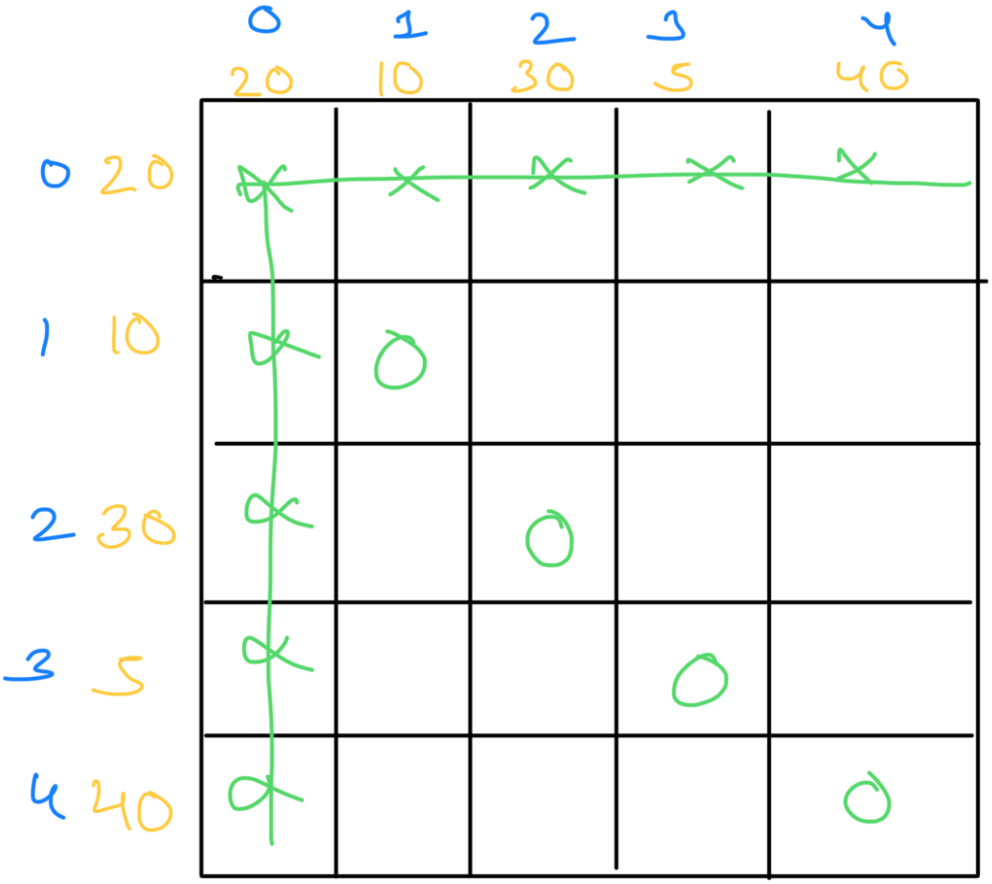
RECURSIVE APPROACH

CODE :

```
def matrixMultiplication(N, arr):
    def findCost(i, j):
        #check for single matrix
        if i == j:
            return 0    # base case

        cost = float('inf') #setting cost to
                            highest value

        for k in range(i, j):

            cur = findcost(i, k) + findcost(k+1, j)
                + arr[i-1] * arr[k] * arr[j]
```

```
            cost = min(cost, cur)
        return cost
    return findCost(1, N-1);
```

TABULATION APPROACH



CODE :-

```python
def matrixMultiplication(N, arr):
    dp = [[0]*N for _ in range(N)]
    for l in range(1,N+1):
        for i in range(1,N-l+1):
            j = i+l - 1
            if i==j:dp[i][j] = 0
            else:
                dp[i][j] = float('inf')
                for k in range(i,j):
                    dp[i][j] = min(dp[i][j],dp[i][k]+dp[k+1][j]+arr[i-1]*arr[k]*arr[j])
    return dp[1][N-1]
```

Similar to memoization Technique

QUESTION - MAXIMUM SOBARRAY

Kadane's Algorithm

CODE :

```
def maxSubArray(nums):
    max_sum = float('-inf')
    curr_sum = 0
    for i in range(len(nums)):
        currsum += nums[i]
        max_sum = max(currsum,
                      max_sum)
        if curr_sum < 0:
            curr_sum = 0
    return max_sum
```

It seems like a greedy approach but it is considered as D.P.

QUESTION : 152. Maximum Product Subarray

```python
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        # Handle empty input
        if not nums:
            return 0

        # Initialize variables with the first element
        max_so_far = nums[0]  # Max product ending at current position
        min_so_far = nums[0]  # Min product ending at current position
        result = max_so_far   # Overall max product found

        # Iterate through the array starting from the second element
        for i in range(1, len(nums)):
            curr = nums[i]

            # Calculate the new max product ending at current position
            # It can be either the current number itself, or its product with
            # the previous max or min (to account for negative numbers)
            temp_max = max(curr, max_so_far * curr, min_so_far * curr)

            # Calculate the new min product ending at current position
            # Similar logic as max, but using min function
            min_so_far = min(curr, max_so_far * curr, min_so_far * curr)

            # Update max_so_far
            # We use a temp variable because we need the old max_so_far value
            # to calculate min_so_far
            max_so_far = temp_max

            # Update the overall max product if necessary
            result = max(result, max_so_far)

        return result
```