```
def word_break(s, word_dict):
    word_set = set(word_dict)
    dp = [False] * (len(s) + 1)
    dp[0] = True
     for i in range(1, len(s) + 1):
        for j in range(i):
            if dp[j] and s[j:i] in word_set:
                dp[i] = True
                break
   return dp[-1]
s = "leetcode"
word_dict = ["leet", "code"]
print(f"Can the word '{s}' be segmented: {word_break(s, word_dict)}")


def word_trap(board, word):
    def dfs(board, word, i, j, k):
        if k == len(word):
            return True
        if i < 0 or i >= len(board) or j < 0 or j >= len(board[0]) or board[i][j] != word[k]:
            return False
 temp = board[i][j]
        board[i][j] = '#'
        found = (dfs(board, word, i + 1, j, k + 1) or
                 dfs(board, word, i - 1, j, k + 1) or
                 dfs(board, word, i, j + 1, k + 1) or
                 dfs(board, word, i, j - 1, k + 1))
        board[i][j] = temp
        return found


    for i in range(len(board)):
        for j in range(len(board[0])):
            if dfs(board, word, i, j, 0):
                return True
    return False
board = [
    ['A', 'B', 'C', 'E'],
    ['S', 'F', 'C', 'S'],
    ['A', 'D', 'E', 'E']
]
word = "ABCCED"
print(f"Can the word '{word}' be found in the grid: {word_trap(board, word)}")


def optimal_bst(keys, freq, n):
    cost = [[0 for x in range(n)] for y in range(n)]


    for i in range(n):
        cost[i][i] = freq[i]


    for L in range(2, n+1):
        for i in range(n-L+1):
            j = i + L - 1
            cost[i][j] = float("Inf")
                         for r in range(i, j+1):
                c = ((cost[i][r-1] if r > i else 0) +
                     (cost[r+1][j] if r < j else 0) +
                     sum(freq[i:j+1]))
                if c < cost[i][j]:
                    cost[i][j] = c
        return cost[0][n-1]
keys = [10, 12, 20]
freq = [34, 8, 50]
n = len(keys)
```

```
print(f"Cost of the Optimal Binary Search Tree is {optimal_bst(keys, freq, n)}")


class ListNode:
    def __init__(self, value=0, next=None):
        self.value = value
        self.next = next


def has_cycle(head):
    slow = head
    fast = head

    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

        if slow == fast:
            return True
    return False
head = ListNode(1)
head.next = ListNode(2)
head.next.next = ListNode(3)
head.next.next.next = ListNode(4)
head.next.next.next.next = head.next  # Creating a cycle
print(f"Does the linked list have a cycle: {has_cycle(head)}")
```