Day 41 / 100 :

Topic - Backtracking, Array

1 Problem statement: Word Break (Medium)

Given a string s and a dictionary of strings wordDict, return true if s can be segmented into a space-separated sequence of one or more dictionary words.
Note that the same word in the dictionary may be reused multiple times in the segmentation.

Example 1:
Input: s = "leetcode", wordDict = ["leet","code"]
Output: true
Explanation: Return true because "leetcode" can be segmented as "leet code".

Example 2:
Input: s = "applepenapple", wordDict = ["apple","pen"]
Output: true
Explanation: Return true because "applepenapple" can be segmented as "apple pen apple".
Note that you are allowed to reuse a dictionary word.

Solutions :
Approach 1 - Recursion + LinkedList

Intuition:
We can solve this question using Multiple Approaches. (Here I have explained all the possible solutions of this problem).

1. Solved using String + Backtracking + Hash Table.
2. Solved using String + DP(Memorisation) + Hash Table.
3. Solved using String + DP(Tabulation) + Hash Table.

Approach:
We can easily understand the all the approaches by seeing the code, which is easy to understand with comments.

Complexity:

- Time complexity:

Time complexity is given in code comment.

- Space complexity:

Space complexity is given in code comment.

```cpp
/************************************* Approach 1 First Code
*************************************/

class Solution {
private:
    bool wordBreak(string s, unordered_set<string> &set){
        if(s.size() == 0){
            return true;
        }
        for(int i=0; i<s.size(); i++){
            if(set.count(s.substr(0, i+1)) && wordBreak(s.substr(i+1), set)){
                return true;
            }
        }
        return false;
    }
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        unordered_set<string> set(wordDict.begin(), wordDict.end());
        return wordBreak(s, set);
    }
};




/*

    Time Complexity : O(2^N), Given a string of length N, there are N+1 ways to
split it into two parts. At each
    step, we have a choice: to split or not to split. In the worse case, when all
choices are to be checked, that
    results in O(2^N).

    Space Complexity : O(N), The depth of the recursion tree can go upto N.

    Solved using String + Backtracking + Hash Table.

*/


/************************************* Approach 1 Second Code
```

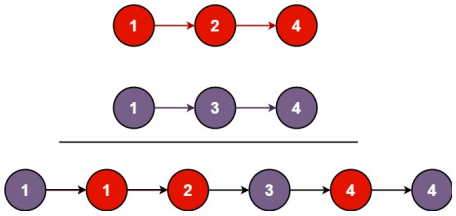
```
*************************************/

class Solution {
private:
    bool wordBreak(string s, unordered_set<string> &set, int start){
        if(start == s.size()){
            return true;
        }
        for(int i=start; i<s.size(); i++){
            if(set.count(s.substr(start, i+1-start)) && wordBreak(s, set, i+1)){
                return true;
            }
        }
        return false;
    }
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        unordered_set<string> set(wordDict.begin(), wordDict.end());
        return wordBreak(s, set, 0);
    }};
/*
    Time Complexity : O(N^3), Size of recursion tree can go up to N^2.

    Space Complexity : O(N), The depth of the recursion tree can go upto N.

    Solved using String + DP(Memoisation) + Hash Table.
*/

/************************************* Approach 2
*************************************/

class Solution {
private:
    bool wordBreak(string s, unordered_set<string> &set, vector<int> &memo, int
start){
        if(start == s.size()){
            return true;
        }
        if(memo[start] != -1){
            return memo[start];
        }
        for(int i=start; i<s.size(); i++){
            if(set.count(s.substr(start, i+1-start)) && wordBreak(s, set, memo,
i+1)){
                memo[start] = true;
                return true;
            }
```

```
        }
        return memo[start] = false;
    }
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        vector<int> memo(s.size(), -1);
        unordered_set<string> set(wordDict.begin(), wordDict.end());
        return wordBreak(s, set, memo, 0);
    }
};
/*
    Time Complexity : O(N^3), There are two nested loops, and substring computation
at each iteration. Overall
    that results in O(N^3) time complexity.

    Space Complexity : O(N), Length of dp array is N+1.

    Solved using String + DP(Tabulation) + Hash Table.

*/


/**************************************** Approach 3
*****************************************/

class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        vector<bool> dp(s.size()+1, 0);
        dp[0] = true;
        unordered_set<string> set(wordDict.begin(), wordDict.end());
        for(int i=1; i<=s.size(); i++){
            for(int j=0; j<i; j++){
                if(dp[j] && set.count(s.substr(j, i-j))){
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[s.size()];
    }
};
```

2 Problem statement: **Merge Two Sorted List** (Medium)

You are given the heads of two sorted linked lists list1 and list2.
Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists.
Return the head of the merged linked list.

Example 1:



Input: list1 = [1,2,4], list2 = [1,3,4]
Output: [1,1,2,3,4,4]

Example 2:
Input: list1 = [], list2 = []
Output: []

Example 3:
Input: list1 = [], list2 = [0]
Output: [0]

Solutions :
Approach 1 - Recursion + LinkedList

Explanation:

- Maintain a head and a tail pointer on the merged linked list.
- Then choose the head of the merged linked list by comparing the first node of both linked lists.
- For all subsequent nodes in both lists, you choose the smaller current node and link it to the tail of the merged list, and moving the current pointer of that list one step forward.
- You keep doing this while there are some remaining elements in both the lists.
- If there are still some elements in only one of the lists, you link this remaining list to the tail of the merged list.
- Initially, the merged linked list is NULL.
- Compare the value of the first two nodes, and make the node with the smaller value the head node of the merged linked list.
- Since it's the first and only node in the merged list, it will also be the tail.
- Then move head1 one step forward.

Complexity:
Time Complexity O(n+m)
Space Complexity O(n+m) this is auxiliary stack space due to recursion.

```
class Solution {
public:
      ListNode* mergeTwoLists(ListNode* l1, ListNode* l2)
  {
            // if list1 happen to be NULL
            // we will simply return list2.
            if(l1 == NULL)
        {
                  return l2;
            }

            // if list2 happen to be NULL
            // we will simply return list1.
            if(l2 == NULL)
        {
                  return l1;
            }

            // if value pointend by l1 pointer is less than equal to value pointed
by l2 pointer
            // we wall call recursively l1 -> next and whole l2 list.
            if(l1 -> val <= l2 -> val)
        {
                  l1 -> next = mergeTwoLists(l1 -> next, l2);
                  return l1;
            }
            // we will call recursive l1 whole list and l2 -> next
            else
        {
                  l2 -> next = mergeTwoLists(l1, l2 -> next);
                  return l2;
            }
      }
};
```